Recognising the Design Decisions in Prolog Programs as a Prelude to Critiquing

Diana Bental

Ph.D.

University of Edinburgh

1993

Recognising the Design Decisions in Prolog Programs as a Prelude to Critiquing

Diana Bental

Ph.D.

University of Edinburgh

1993

Abstract

This thesis presents an approach by which an automated teaching system can analyse the design of novices' Prolog programs for tutorial critiquing. Existing methodologies for tutorial analysis of programs focus on the kind of small programming examples that are used only in the early stages of teaching. If an automated teaching system is to be widely useful, it must cover a substantial amount of the teaching syllabus, and a critiquing system must be able to analyse and critique programs written during the later stages of the syllabus.

The work is motivated by a study of students' Prolog programs which were written as assessed exercises towards the end of their course. These programs all work (in some sense), yet they reveal a wide range of design flaws (bodges) for which some form of tutoring would be useful. They present problems for any automated analysis in terms of the size of the programs, the number of individual decisions that must be made to create each program and the range of correct and incorrect decisions that may be made in each case.

This study identifies two areas in the analysis of students' program in which further work is needed. Existing work has focussed only on the design and implementation decisions that relate closely to the programming language. That is not sufficient for these slightly more advanced programs, for which decisions in the problem domain must also be recognised. Existing work has focussed on the different ways to implement code, but in these programs the students also make decisions about which data structures are to be used. These decisions must also be part of an analysis.

The thesis provides an approach which represents both decisions in the domain of the problem being solved and decisions about how to implement them in Prolog. Decisions in the problem domain are represented by tasks (for code) and by domain objects (for data structures). Decisions that are specific to the Prolog implementation are represented by prototypes which encapsulate standard programming techniques (for code) and by a polymorphic data type language (for data structures). Issues in devising these representations are discussed.

An analysis–by–synthesis approach is used for code recognition. This is augmented by a procedure called "clausal split" which isolates novel or poorly designed parts of an implementation. Following an incomplete analysis of the program by synthesis, the results of this analysis provide the basis for making inferences about the parts of the program that have not been understood. For analysing data structures, a type inference mechanism is combined with inference about the parts of domain objects. Inferred data type information is also used to limit search, both for synthesis and analysis.

An architecture using this approach has been implemented. The success of the architecture is assessed on student's programs. From this assessment it is clear that much further work remains to be done, but the results are hopeful.

Declaration

I declare that I composed this thesis entirely myself and that it describes my own research.

Diana Susan Bental
Edinburgh
March 16, 1994

Some of the material reported in this thesis has been published elsewhere:

- A description of the clausal split algorithm, in the Proceedings of the First AAAI Workshop on Artificial Intelligence and Automated Program Understanding, San Jose, US, 1992.

- A description of the analysis of data structures, in the Proceedings of the World Conference on Artificial Intelligence and Education, Edinburgh, UK, 1993.

Acknowledgements

I would like to thank Peter Ross, Paul Brna and Tim Smithers for supervising this work. I would especially like to thank Paul Brna for his encouragement and willingness to the discuss even the most intricate technical details of my work while still keeping me on a coherent track.

I would like to thank the many people who have read and commented on drafts of this thesis. Paul, Peter and Graeme Ritchie provided many insightful comments on drafts. Thanks for comments are due also to Greg Michaelson, Tim Smithers, Helen Lowe, Sheila Rock, Lex Holt and Susan Bull.

Thanks are due to members of the Knowledge Based Systems Group for discussions and ideas, especially Andy Bowles, Wamberto Vasconcelas and Maria Vargas Vera. I am indebted to Dave Berry for proposing polymorphic type languages as a potential solution to the problem of representing data structures.

Students of the Prolog module in the 1988-89 and 1990-91 MSc courses provided me with sample Prolog programs for study and for assessment of the system. Thanks to them and to Dave Robertson for organising my access to these programs.

I would like to thank the many friends and colleagues who have made living in Edinburgh and working in the Department of Artificial Intelligence an interesting and rewarding experience.

Heartfelt thanks to my flatmates Sheila and Vashti, who took on housework and tolerance beyond the call of duty while I wrote up. And thanks especially to Dave, whose love and encouragement made many things possible.

This work had been supported by a studentship from the Science and Engineering Research Council. I would also like to thank the Artificial Intelligence Applications Institute at the University of Edinburgh for providing part-time employment during much of this period and for funding my attendance at the AAAI 1993 Conference.

Table of Contents

Chapter 1

Introduction

1.1 Overview

Programming is a demanding intellectual task which requires a variety of problem-solving skills, and novices have a variety of problems in learning them. Computer programming is generally considered to be a good field for automated tutoring using Artificial Intelligence (AI) techniques because the knowledge and problem-solving strategies it uses are specific to a domain rather than involving broad general knowledge about the world (Wenger, 1987).

Our work is motivated by the study of *bodges*, related to the study of *bugs*. Bugs typically cause a program to misbehave in some way, to exhibit behaviour that is counter to the program's functional specification. Bodges enable a program to fulfill its functional specification, but they do this in such a way that they violate the constraints that would result in a good design. Bodges create programs that are unreadable, inefficient, unmaintainable and unreliable. We argue that bodging is a necessary part of learning to program and that therefore tutoring systems which prevent bodging are only of limited value, whereas a system which could detect and criticise bodges would be an important contribution for automatically teaching students to produce better designed programs in future.

We have investigated a study set of programs produced as exercises by students half-way through a Prolog course. The exercise is to produce a player for a game of noughts and crosses. All of the programs studied fulfill the functional specification as given by the lecturer — that is, they contain errors that are bodges rather than bugs. We characterise the students' task in terms of the design decisions that are required and how those decisions are related to each other. We distinguish between strategic, algorithmic and implementation decisions, and note the importance of choosing appropriate data structures.

In order to recognise bodges we must model the design decisions that the student makes in writing as program to solve a problem, the relationships between them and the constraints that they impose on later design decisions.

There are specific problems of trying to analyse and critique programs of this size and complexity. These programs involve over 100 lines of code and many sub-programs. They require a large number of different decisions which require different kinds of knowledge. The program specification is incomplete and the student must make decisions about the behaviour of the program. The specification omits many details about the design. For example, the program specification states that some kinds of objects are to be represented by data structures, but the student decides which data structures are used to implement them. The programs may include any feature of the programming language rather than a restricted subset of the language. By this stage the students have been taught many aspects of the programming language and they may use them well or badly.

Most of the existing research effort has gone into tutoring rank novices. There are good reasons for that. The programs are shorter, single figures or tens of lines, requiring few sub-programs. The data structures are specified in the problem specification. There are only a few good solutions, so it is possible to enumerate them and to enumerate the most common diversions from those solutions. Much effort has gone into recognising variants of the good solutions, into distinguishing between good variants and bad (buggy) ones, and into correcting these bugs (Looi, 1988b; Murray, 1988; Johnson, 1990).

Intelligent debugging requires that we understand not only what the student has done but also what the student intends to do (Johnson, 1990). This falls into two parts: working out what purpose a particular piece of code is meant to fulfill, and working out what algorithm a student is trying to write so as to fulfill that purpose, even if the student has failed to write it properly (Murray, 1988).

Existing work on intelligent debugging for novices concentrates on the latter and presupposes the former. In programming exercises for novices, the purpose of the program as a whole is known, because it is specified by the exercise. Sub-programs are a little more difficult, but programs which rank novices write do not have many sub-programs and their purposes can be enumerated quite easily. For these problems the purpose of code only really becomes difficult to determine when looking at single lines of code. It is not possible to tell the purpose of each line of code individually and a single line of code may fulfill more than one purpose. But once a piece of code has been mapped onto a plan or schema then its purpose is plain. In small programs the range of possible purposes for a piece of code is very constrained, and even if the exact purpose of each line of the code is not known in advance it is possible to presuppose and enumerate the purpose of bigger pieces. This assumption falls apart in larger programs in which the same sub-program may be called in different contexts.

We have implemented an architecture in which design decisions about data structures and algorithms can be represented and recognised. We have represented a plan structure of tasks and sub-tasks that must be performed in order to fulfill the noughts and crosses specification. The task structure also describes the data objects on which those tasks are performed. Linked to the task structure are prototypical predicates, i.e. schematic descriptions of the typical correct ways to implement such tasks in Prolog. From the task structure and the prototypical predicates the system automatically synthesises canonical code that implements different versions of noughts and crosses, and it matches the generated code against the student's code. Transformation rules allow us to generate some minor variants of the canonical code and so match a wider range of programs.

Prototypical predicates offer a way to represent a wide variety of different Prolog programming techniques. Some aspects of this approach to generating code have already been implemented, e.g. Sterling and Lakhotia's procedures for clausal join. We describe the application of this representation to the recognition of programming techniques, including our design and implementation of *clausal split*, the inverse of clausal join. We use the clausal split to identify components of predicates so that components can be matched even if the whole predicate cannot.

Students can produce a wide range of code variants, correct and bodged. We do not know in which order an automated system should apply transformations that derive the student's code from the canonical versions. We expect that the student will have fulfilled some of the tasks in ways that any given automated system could not recognise at all. An automated system needs to hypothesise that certain parts of the code fulfill particular tasks, and to confirm or change these hypotheses as the analysis proceeds. The system needs to combine information from different sources. As recognition proceeds, the system notes that certain functions appear to have been fulfilled and posts these as constraints on the likely purposes of other, unrecognised code.

Code is generated for matching only "as needed", and information is gathered from different sources and combined in a forward-driven manner. Code generation from schematics, information gathering and integration of information are performed in a fixed order. A more flexible system would include defeasible reasoning and constraint satisfaction, and be supported by a truth maintenance system.

The analysis system has been provided with a knowledge base to enable it to identify one part of the noughts and crosses program. The system has been assessed on student's programs using this knowledge base.

The behavioural analysis using type inference appears to be rather more robust than the structural analysis. This suggests that the behavioural analysis should be given a high priority for the task of understanding the purpose of code.

We conclude that a tool for detecting and critiquing bodges is useful for teaching students to build well-designed programs. We find that by accepting an incomplete and defeasible analysis we can take some steps towards building such a tool.

In the context of building teaching tools, the success of our system is limited by the power of existing analysis techniques such as polymorphic type inference. Considerable work further work is also needed to identify a full set of programming techniques in Prolog. The system would be greatly improved by a more explicit and flexible control structure.

If we broaden the context of our work to tools which support software design in general, further serious problems must be solved. In a tutorial setting, the analysis program has a large amount of knowledge about the problem to be solved and the best ways to solve it. A general design assistant would need to work with much less knowledge about the specific problem. It would also need to accept corrections from the designer and learn new implementation techniques in a co-operative manner.

1.2 Contribution

We have studied working programs of more that 100 lines of code written by novice programmers. The study shows that when students write a program that consists of many parts, even when it is a working program that fulfills the problem specification, parts of their code still contain many errors. To correct these errors requires an understanding not only of general principals of programming style in Prolog but also an understanding of the purpose that each piece of code is intended to fulfill within the program. A debugger or style critiquer that could understand and tutor these errors would be a useful teaching aid, but existing intelligent debuggers for novice programs have not been scaled up to deal with programs of this size.

We have implemented an automated analysis that identifies both the purpose of the code within the domain of the problem that the student is trying to solve, and the programming techniques that the student is using in the Prolog programming language to implement their solution. This analysis has three parts:

(1) An analysis of data structures that uses polymorphic types to describe abstractly the data structures that the student has used, and a type inference algorithm to infer from the program which Prolog data structures have been used. The type inference algorithm recognises decisions about data structures in terms of the Prolog programming language. The type language and type inference algorithm are supplemented with heuristics to identify these data structures with the objects in the problem domain that the student wishes to represent

(2) An analysis–by–synthesis approach as used by (Johnson, 1990), in which the student's code is identified with code that is synthesised for particular tasks from prototypical predicates. Prototypical predicates represent programming techniques, and implementations of the same task may be synthesised in many ways.

This thesis does not deal with the problem of matching variants of code. Approaches to this problem using heuristics, formal proofs of equivalence and abstraction have been explored in (Looi, 1988b; Murray, 1988; Johnson, 1990; Wills, 1990).

The strong connection between goals and plans in (Johnson, 1990) in which each goal contains an explicit declaration of the plans that may be used to complete that goal is weakened, so that prototypical predicates are chosen to fulfill a task if they have appropriate properties. Type inference is further used to check that appropriate prototypes are applied to each task and that the synthesised code is correct. This check does not remove all incorrect implementations but it is a step away from demanding that anyone wishing to build such a system must specify exactly which implementations go with which tasks and a step towards a formal check on whether the system builder has told the system to recognise as correct code that is actually wrong.

(3) When the analysis-by-synthesis approach fails on a poor or novel implementation, the student's code is teased apart into components using a process called clausal split. These components can be matched to confirm hypotheses about the purpose of the code and to isolate the programming techniques that the student has misused.

Our work reveals two major unsolved problems in scaling up intelligent debugging to deal with large novice programs. The first problem is that of representing abstract programming knowledge that is neither specific to the problem domain not to the programming language. The second problem is that of the many variants of a program that a student might reasonably implement. The enormous range of idiosyncratic problem-solving strategies and implementations that students may try mean that any critiquing system for a specific programming task requires that an immense quantity of knowledge must be built in to the system, which is a severe task for anyone wanting to create such a system for use in teaching. A solution to the first problem is likely to help with the second, by imposing an intermediate level of abstraction.

1.3 Structure of the Thesis

The remainder of the thesis is structured as follows:

Chapter 2 looks at existing automated systems which teach computer programming and at the aspects of programming that they support. Intention-based approaches to recognising and tutoring students' programming errors have been successful with small exercises for novices, and they might usefully be extended to the larger and less closely specified programming exercises assigned to more advanced students.

Chapter 3 describes a study of the programming errors that Prolog students made in a programming exercise. This chapter discusses the different aspects

of software design that students need to know about in order to correct those errors. The students are beyond the initial novice state that is supported by most automated tutoring systems. A critique of their programs depends on understanding what each student is trying to achieve, the programming methods that the student is trying to use and also the purpose for which the code is intended. This study motivates our need for automated tutoring and automated recognition within loosely specified programs of several hundred lines of code.

Chapters 4 and 5 describe our approach to recognising design decisions in students' Prolog programs. We have built an architecture in which design decisions about data structures and algorithms can be represented and recognised. Chapter 4 describes and motivates the declarative knowledge structures that are used in that architecture, while Chapter 5 describes how these knowledge structures are manipulated.

Chapter 6 demonstrates how the system operates in detail on example programs. It also assesses how successfully the system performs in recognising tasks and techniques, data objects and their implementations within a series of students' programs. The assessment is described on familiar and unfamiliar programs.

Chapter 7 describes our conclusions and the implications of the work for automated tutoring of computer programming.

Background data for the thesis are included in the Appendixes. Appendix A gives the examples of students' programming errors and bodges that we found in the study described in Chapter 3. Appendix B is a listing of the simplified noughts and crosses program that is used in the thesis as an illustrative example. Appendix C is a listing of a similar program in which one of the predicates has been bodged, and is used to illustrate clausal split. The analysis of both these programs is presented in detail in Chapter 6 Sections 6.3 and 6.4. Appendix D shows 16 students' implementations of a single task, taken from the study

set described in Chapter 3 and used in the evaluation in Chapter 6 Section 6.5. Appendix E shows the task and prototype definitions that we devised for parts of the noughts and crosses program and used for the analyses described in Chapter 6.

Chapter 2

Automated Support for Novice Programmers

2.1 Introduction

Computer programming requires a collection of different skills and knowledge, all of which novices must eventually acquire. These skills include:

1. writing syntactically correct programs;
2. writing semantically correct programs that use the computational model properly;
3. debugging;
4. using common programming idioms and solution methods;
5. writing abstract problem solutions and mapping the abstract solutions onto the programming language;
6. writing programs that are easy to read, reuse and maintain;
7. building larger-scale programs that are modular.

A variety of automated tutoring systems for computer programming have been implemented, and a variety of approaches explored. Different automated tutoring systems support different skills and different aspects of programming knowledge.

Providing intelligent automated support for novice programmers is not easy. In spite of a large body of research, only a very few AI-based systems have been used in practice to teach programming. The most successful existing systems for novices focus on teaching the syntax and semantics of the programming languages, but they do not support the acquisition of other programming skills and knowledge. The automated tutoring of some skills has not been explored even in current research prototypes.

In this chapter, we consider the knowledge and skills that novice programmers must acquire in order to write programs. We investigate the role of bugs and programming errors in learning to program. We describe two automated approaches to teaching computer programming and outline the automated tutorial systems that have been implemented, the aspects of programming that they teach and the methods they use. We explore the difficulties of detecting and correcting students' programming errors, and we assess the success and limitations of different automated approaches to teaching programming.

Three important criteria for the success of a tutorial system are the range of programming exercises that it teaches, the size of the exercises, and the variety of implementations and programming strategies that it can handle. A successful tutorial system must be capable of tutoring a large number of programming tasks. Small programs are sufficient for beginners, but more advanced students will need to write larger programs. It must be able to deal with the various different algorithms that students attempt and the many variants of those algorithms they implement. A tutorial system must be able to correct the visible errors that students have made and to help the student correct the inner misconceptions that lead to those errors. A tutorial system needs to correct both errors that lead to programs that do not work and also errors that lead to working but badly designed (bodged) programs.

This thesis considers mainly the identification of design decisions. We do not consider the question of deciding which decisions are erroneous and how to correct misconceptions. We then consider automated intelligent support to experienced

programmers who are writing production software. These systems do offer some support for other aspects of programming, and so we look at what methods they use and whether they could be applied to teaching novices, and we assess how successful existing systems are at understanding a range of different programs and at dealing with bugs.

2.2 Programming, Bugs and Bodges

2.2.1 Misconceptions, Bugs and Bodges

Novices have difficulties in learning the skills required for programming, and these difficulties cause students to produce buggy programs. Bugs suggest the existence of gaps or misunderstandings in the student's knowledge, which could be remedied by tutoring. Bugs also offer an opportunity to provide that tutoring in the context of the student's own work.

Studies have identified bugs in computer programs that arise from many different sources. Many bugs that confuse novices deeply may be due to memory lapses (Reiser *et al*, 1985) or typographical errors (Looi, 1988b). Other bugs are due to misconceptions about how to program in that language: misunderstandings about the primitives and execution behaviour of the language (Taylor, 1990; Fung *et al*, 1990), or inexperience in using and combining primitives (Spohrer & Soloway, 1986). Still others are due to general inexperience in algorithmic or computer-based problem solving (Adelson & Soloway, 1985; Adelson *et al*, 1985) or to failure to understand the problem specification properly (Spohrer & Soloway, 1986).

It can be very difficult for novices to understand what a computer is doing with their instructions or programs. Novices construct their own incorrect explanations for the effects of their programming actions, which they then stick to loyally in spite of conflicting evidence (Carroll & Aaronson, 1988). This is especially a

problem in teaching the Prolog programming language, which has a complex and unusual computational model (Taylor, 1990).

Anderson's ACT* theory predicts and models students' memory and learning errors and is the basis for the LISP Tutor (Anderson *et al*, 1984). The LISP Tutor represents correct solution methods and some common errors. All deviations from the correct path are treated as errors which are due to the students' having forgotten or failed to acquire some programming concept. ACT* theory is a general model of students' learning behaviour.

MARCEL (Spohrer & Soloway, 1989) is a cognitive model of how students write programs and the bugs that they create. By contrast with the LISP Tutor's general student model, MARCEL's model accounts for the differences between programs written by different students. MARCEL proposes a model of programming in which students cycle through a sequence: generating a solution, testing it and debugging it. Students reach impasses and try to repair them using both programming and non-programming knowledge. Students criticise and correct their own programs according to the design criteria they have learned. Different repairs to impasses create different versions of the code. In this model, buggy programs arise from incomplete sequences of repairs to an impasse.

Some programming errors lead to programs that do not run at all or that fail to fulfill the behaviour required by the problem specification. These programs are certainly buggy. Other errors lead to programs that do run but do not fulfill criteria for good programming style. Chapter 3 describes a study of students' Prolog programs which fulfilled the program specification, and yet still contain many of these latter kind of programming errors. We call these errors *bodges*. In the MARCEL model of individual design, we might account for bodged programs by describing them not as incomplete sequences of repairs to impasses but as inappropriate repairs to impasses. Bodges result in programs that are successful in terms of the general problem specification but unsuccessful in terms of criteria for good program design. Bodged programs are unreliable, inefficient and idiosyn-

cratic. They are difficult to read and to re-use (Joni & Soloway, 1986). Bodges have been found in other studies of novice programs (Spohrer *et al*, 1985).

This section has explained why we want an automated system to recognise and critique programming errors in students' programs. The following section describes the approaches to automated tutoring of computer programming that have been taken so far.

2.3 Two Approaches to Automated Teaching

Intelligent Tutoring Systems have been built to teach subjects in which the amount of factual knowledge is relatively small and the problem-solving methods are easily formalised. These subjects include arithmetic (Burton & Brown, 1981), specialised medical domains (Clancey, 1987; Hasling *et al*, 1983; Miller, 1984; Swartout, 1983) and the diagnosis of electronic devices (Brown *et al*, 1981). Tutoring systems have also been developed to support the more formal aspects of computer programming (Miller, 1981; Murray, 1987; Reiser *et al*, 1989).

Automated systems for computer programming have followed one of two contrasting approaches to teaching novice programmers. One approach allows students to create buggy programs using a normal computing environment and then uses *tutorial debugging* to identify errors in a student's completed program and propose corrections and improvements to their programs (Johnson & Soloway, 1984; McCalla & Greer, 1988; Murray, 1987; Looi, 1988a). The second approach provides a highly interactive *tutorial environment* for writing programs which monitors and guides students closely as they write their programs so as to restrict the errors that students can make (Reiser *et al*, 1985; Bonar & Cunningham, 1988). These are two extremes and tutorial systems that combine some features of both could be devised, but in reality there are few systems that do so and we describe the approaches separately.

The guided approach to creating error-free programs has proved to be successful in the initial stages of learning to program (Corbett *et al*, 1993). It also has practical advantages that make such systems easier to build for real programming courses. Nevertheless, the freedom to create and correct buggy programs is important to the more advanced stages of learning and we have developed an architecture for tutorial debugging rather than a tutorial environment.

These two approaches are described in more detail in the following sections (Sections 2.4 and 2.5).

2.4 Tutorial Debugging for Novice Programmers

In this section, we discuss tutorial debugging systems which take a novice's computer program which contains some bugs and propose corrections to the program.

One approach to debugging is via the tracers that are supplied with programming languages. However, these are thought to be of most use to experienced programmers. A program tracer for novices can present the computational model of the language clearly and make it easier for novices to debug their own programs (Eisenstadt & Brayshaw, 1986). Nevertheless, tracers without knowledge of the task being undertaken put a strong burden of explanation onto the user. Tracers present the behaviour of the program to the user, leaving the user to work out how the trace relates to the behaviour they want from their program.

Diagnosis and tutoring both require that the tutor understands what the student intended the program to do and what constructs the student was trying to use. Semantic and teleological errors are difficult to diagnose without an understanding of the task that is being implemented.

> ...in order to reliably diagnose as near to the complete range of semantic and logical errors as possible, a debugging system must understand the programmer's intentions. A program is a designed artifact; as such, its design must be taken into account when analysing it for bugs. (Johnson, 1990)

Task-specific debugging uses knowledge about the particular task the student is programming and the ways of solving that problem, whereas general debugging relies only on general knowledge about programming.

Tutoring systems that analyse novice's teaching exercises can include representations of the desired behaviour of the code, the most common algorithms used to solve the exercises and even the most common bugs. This may be as simple as asking the teacher to specify the program's expected behaviour (Barr *et al*, 1976). More complex systems require some explicit representation of the possible implementations, and use transformation rules or formal proofs to show that the student's program is equivalent to a reference implementation and to identify bugs (Johnson, 1990; Murray, 1987; Looi, 1988a).

Some debugging systems rely on general principles of programming and do not use information about the particular task that is being implemented. The LISP Critic (Fischer, 1987) applies program transformation rules to suggest local improvements in program code which would make programs more readable or efficient. PHENARETE (Wertz, 1982) and Elsom-Cook and du Boulay's Pascal syntax checker (Elsom-Cook & duBoulay, 1988) are among the few programs which can transform code with syntax errors into correct code. These systems both rely on knowledge about the correct syntax in the programming language and also on rules about the kinds of syntax errors that novices commonly make. These systems are more general than systems which require task specifications, but they do not know what code or behaviour fulfills the intended task and so they cannot detect and correct teleological bugs that cause the program's behaviour to violate the task specifications. Often there is more than one possible way to correct a syntax error. Some corrections can be proposed purely in terms

of general novices' behaviour (Elsom-Cook & duBoulay, 1988), but some corrections appear to be specific to particular programming tasks and in other tasks they would lead to programs that were syntactically correct but incorrect for the task (Wertz, 1982). Our focus is on novices at a more advanced stage of learning who are able to write syntactically correct programs and correct their own syntax errors, and so we deal with teleological errors rather than syntactic ones.

In the remainder of this section, we focus on task-specific tutorial debugging. We describe the use of static debugging, which inspects the form of the student's code, and dynamic debugging which runs the code on particular inputs and inspects the outputs.

2.4.1 Error Detection by Static Analysis

Static analysis of computer programs inspects the form of the computer program, the code itself. The static analyses described in this section compare the form of the student's code with canonical descriptions of programs that solve the exercise. Bugs can be detected when the code differs from the canonical solution. Tutoring systems must also deal with correct variations in programs. There are many ways to write a program correctly, usually too many for every variation to be included explicitly. A tutor must recognise that correct variants of a program are indeed correct.

Different systems use various forms of canonical description, bug detection and dealing with variations. Laura (Adam & Laurent, 1980) uses a graph representation of the algorithm, which is compared to a graph representation that is derived from the student's program. Bug detection arises from the difference between the two. Laura was not able to deal with many correct variations. Proust (Johnson, 1990) uses a plan representation from which a hierarchical goal structure leads down to the actual code. Some bugs are represented by buggy plans, others by heuristic rules. Correct variations are also represented by heuristic rules. Apropos (Looi, 1988b) and Talus (Murray, 1988) use a series of schematic

programs in the same programming language as the novices' programs. Apropos uses heuristics and dynamic analysis for detecting bugs and variations, whereas Talus uses a theorem prover.

Proust: A Plan–Based Approach

The goal of Proust is to debug novice Pascal programs automatically, to recognise the intentions of the programmer even when the programmer has failed to implement their intended algorithm correctly and to take those intentions into account so as to identify the bug accurately and propose the best correction (Johnson, 1990).

Proust uses a knowledge base of programming plans in which each plan is a common way to solve a programming goal. Plans may contain sub–goals which are fulfilled by further plans, resulting in a hierarchy of goals and plans. Plans may contain actual Pascal language primitives, and these primitives act as markers for matching the plans to the code.

Proust uses the goals and plans for an analysis–by–synthesis approach to matching. The problem specification as supplied to Proust consists of some top–level goals. The recogniser tries to identify parts of the student's code with plans that fulfill these goals, which may in turn require that sub–goals are identified. As the program is interpreted, Proust maintains

- an agenda of goals whose implementation in the program has not yet been identified;
- a partial goal decomposition of the problem;
- a record of matches between the plans in the goal decomposition and the student's code.

Search is controlled by a cycle in which a goal is selected from the agenda and then either a plan is selected for that goal and matched to the code, or else the goal is reformulated. The matching of a plan may add new goals to the agenda. Different versions of the goal/plan decomposition are maintained to keep track

of alternative breakdowns into goals and plans. Each version is matched against the student's code and the matches between the plans and the student's code are recorded for that version. If there are several ways a match might succeed then different versions are maintained.

Proust uses heuristics to minimise search when matching plans to code and to minimise ambiguity. Several interpretations of the student's program may be created and so Proust has to decide which interpretation is best. It would be highly inefficient to generate many interpretations and then compare them all, so Proust tries to trap bad partial interpretations and abandon them before they are completed. Heuristics are used to assess interpretations and to compare them with one another.

Our architecture for recognising design decisions uses analysis-by-synthesis to the extent of synthesising code in the target language (Prolog) for matching to the student's code.

Dealing with Variations in the Code

Proust detects bugs in two ways. It stores plans for common incorrect solutions, as well as plans for correct solutions, so one source of bug detection is a match against a plan for an incorrect solution. Another source of error detection arises when a student's code does not exactly match any plan. Proust uses rules to try to explain why the student's code does not match the plan. These rules work out either that the code is an incorrect implementation of the plan or else that the code is an unusual but correct (though possibly badly structured) implementation. These rules are derived empirically from studies of the kinds of errors that novices make. The rules inspect not only the difference between the expected code and the student's implementation but also the context in which the plan exists, including any other errors and misconceptions that the tutor has identified so far.

Talus is a system which deals with variability in code using program verification. Example programs specify correct implementations and provide correct code to

replace buggy student code. Variation between implementations can be coped with by explicit reasoning about computational semantics during debugging.

Talus (Murray, 1987) analyses syntactically correct LISP programs. It contains reference programs which represent correct versions of the program. It begins by using heuristic matching to decide which reference program is the best fit, and then it uses a theorem prover to reason about computational equivalence between the student's program and the reference program. If they are equivalent, then the student's program is considered to be a correct variant of the reference program. If they are not proved to be equivalent, then the student's program is considered to be an incorrect variant. Talus also uses the theorem prover (rather than heuristics) to debug the program. If the student's program cannot be proved equivalent to the reference program, then the steps that are necessary to debug the program are derived automatically from the steps that are needed to repair the proof.

Talus represents:

- the task assignment as given to the student;
- acceptable algorithms to solve the task (represented by frames);
- reference functions that implement the algorithms;
- semantic knowledge about LISP.

Talus first uses heuristic pattern matching to suggest which reference function should be matched, then it uses the theorem prover to prove equivalence and detect bugs. The heuristics offer a quick way to narrow the choice for matching, while the theorem prover offers a slow but accurate and detailed analysis (Murray, 1988).

The stages are:

1. the code is transformed into an easier form for parsing, such that its meaning (input/output behaviour) is preserved;
2. the code is parsed into frames, then matched to algorithm frames and reference functions using heuristic best-first search;

3. for each student's function and matching reference function, Talus performs symbolic evaluation. Talus makes conjectures that program statements are equivalent and tries to prove those conjectures using built-in knowledge about the semantics of LISP and a theorem prover. If the code cannot be proved to be equivalent, then it is assumed to be buggy and is fixed.

4. if the program is buggy then the required repairs to the equivalence proof constitute the required repairs to the program.

Some buggy algorithms are stored explicitly as reference functions. Other bugs are recognised as differences between the student's code and a reference function. Talus detects all bugs in the program, but it may also give "false alarms" which suggest that some code is buggy when it is in fact correct.

We recognise the importance of recognising programs that are small variations of essentially the same implementation, but we do not deal with this problem in our work. Recognition in our architecture is effected by matching code that the system has synthesised against code that the student has written, and the matching is strict. Our system aims to represent only correct implementations, i.e. working versions of the program, but versions may be bodged. Good and bad decisions may be represented explicitly. We expect that large parts of the code will not be recognisable, and so we do not assume if code cannot be recognised then it is necessarily incorrect. Only if a section of code contains an implementation that is believed to perform a particular task and most but not all of the implementation corresponds to a known method for implementing that task, will the system hypothesise that the code corresponds to a bodged implementation.

2.4.2 Error Detection by Dynamic Analysis

The previous two systems have focussed on the form of the code itself, its *static* structure. An alternative approach to understanding and debugging novice programs is actually to run the code on particular test examples. This is known as dynamic analysis (Murray, 1988).

Dynamic analysis looks at the run-time behaviour of a program. Dynamic methods may compare inputs and expected outputs with actual outputs

(Barr *et al*, 1976), they may also include side effects (Miller & Goldstein, 1977), or they may inspect all procedure calls and changes to data structures in the entire program execution history (McCalla *et al*, 1986).

SCENT (McCalla *et al*, 1986; Greer *et al*, 1989) uses information from different types of analysis, including execution traces on specific inputs and error messages. It uses dynamic analysis to identify parts of the code with reference functions and to isolate bugs. The SCENT architecture is based on co-operating entities communicating through a blackboard (McCalla & Greer, 1988; McCalla *et al*, 1986). The components that use the blackboard have expertise in interfacing, students' knowledge, LISP and programming, problem–solving heuristics, and instructional planning (McCalla & Greer, 1988).

SCENT uses the dynamic behaviour of the code to infer its static structure and to localise bugs. It breaks the code down into separate LISP functions and uses their call chart. It then uses the behaviour of each function in order to identify the function with a reference function (McCalla *et al*, 1986). This information is passed to diagnostic modules which localise bugs by finding the part of the code where the behaviour diverges from the behaviour of the reference functions. The process is controlled by modules which run an overall analysis of the solution and know which tests should be run.

Shapiro's PDS (Shapiro, 1983) is an early system that finds bugs in pure Prolog programs by comparing the actual output and the expected output from a given input. PDS performs a behavioural analysis and so we include it here, but it is not suited to novices, nor does it isolate bugs fully automatically. Instead PDS relies on the user acting as an oracle, telling PDS how parts of the program ought to behave on different inputs while PDS steps through the execution. Novices may not know how every part of their programs ought to behave on every combination of inputs that PDS might wish to ask.

Static and dynamic analysis have complementary strengths and limitations. Static analysis may find bugs where none exist, whereas dynamic analysis can miss some bugs. For example, static analysis may not be able to compute that a

student's program is equivalent to a correct reference program, whereas dynamic analysis will not notice code that can never be executed because it is unreachable.

Apropos combines static and dynamic analysis in a system for novices learning Prolog (Looi, 1988a; Looi, 1988b). Reference functions represent each correct algorithm (and some common buggy algorithms, too). Program transformation rules decompose the student's program into a canonical form which is then matched heuristically against the reference functions. A program critic explains discrepancies between the reference program and the student's program. If the code matching fails, dynamic analysis based on PDS (Shapiro, 1983) is used. However, instead of asking the user how the code ought to behave, as in PDS, Apropos works out how the code ought to behave from the reference function. This enables Apropos to recognise that an unfamiliar algorithm is indeed a successful implementation, at least to the extent that test programs cover the possibilities. This combined approach enables Looi to overcome some of the difficulties in tutoring Prolog described in Section 2.2.

2.4.3 Tasks Covered and Bugs Detected

Apropos has been assessed on sets of students' list processing programs intended to: reverse a list, replace the elements of a list, count the atoms in a list, and count the leaves in a structure of nested lists (Looi, 1988b). These programs required sub-tasks, such as appending to lists together. In this assessment, Apropos recognised the algorithms used in 85% of list reversal programs and 95% of the other programs. If the algorithm is identified successfully, Apropos achieves close to full accuracy in bug detection and correction. Only two bugs were misdiagnosed in the list reversal program, and bugs in all the other programs were diagnosed properly. Talus was also assessed on students' solutions to five list processing exercises, similar to those used in Apropos (Murray, 1988). Talus identified more than 90% of algorithms were correctly and detected 90% of bugs. 3% of bugs proved to be false alarms. The success of both Apropos and Talus is limited by whether the full range of algorithms, and especially incor-

rect algorithms, have been captured in the system and supplied with reference functions.

These systems work with a single task or a very small set of tasks. Proust has been evaluated on two programming exercises. The programs analysed by Talus and Apropos involve sub-programs, whereas the programs analysed by Proust do not. The most complex task represented in Apropos, sorting a list, was not fully evaluated and only a few of the possible algorithms were represented. The most complex programs on which Talus was evaluated created a list of all singleton elements in a nested list structure, although algorithms for bubble sorting a list were also prepared. No full evaluation of SCENT has been published, but it has been applied to simple problems in recursive list processing.

Proust, Apropos and Talus can claim some success at this level of complexity, but broadening their scope to larger programs with many more sub-tasks and a greater range of algorithms is a demanding task.

2.4.4 Scaling Up to Classroom Use

Automated programming tutors often represent both correct and common incorrect plans, together with rules or transformations that account for deviations from either sort of plan. An immense effort is required to create such a system for even a single programming exercise. For example in Apropos even the simplest exercise, reversing a list, required that three different algorithms were represented, plus an algorithm for the sub-task of appending two lists. Furthermore, each algorithm in Apropos can include several variations, correct and incorrect, which must be represented explicitly (Looi, 1988b).

Extensive studies are needed to identify all the bad plans that novices create for each exercise (Spohrer *et al*, 1985). This helps to account for the lack of such systems available for use in teaching. Parsimony indicates that it would be better to represent correct plans and then reason about how the student's actions vary from the correct plans (Stevens *et al*, 1981; Brecht & Jones, 1988).

But empirical results suggest that for all but the most trivial tasks, successful intention-based debugging depends on explicitly representing buggy algorithms as well as correct ones (Murray, 1988; Looi, 1988b).

A problem in analysis-by-synthesis is how the analysis system knows whether the code that is synthesised is correct or buggy. In Proust, only certain pre-specified plans may be applied to each goal, and each plan is labelled as either a correct or a buggy way to fulfill that particular goal. The plan selection and labelling of plans as correct or buggy must be done by the system builder. The system itself cannot check whether the system builder has supplied a valid specification of a plan to fulfill a goal. Our system improves this by not forcing the system builder to specify in advance which plans are to be used to fulfill which goals. Instead, plans are selected dynamically to fulfill goals. Correctness cannot be guaranteed, but the synthesised code is checked for the consistent handling of data types.

Proust does not distinguish between different kinds of plans, plans that are specific to the task being implemented or plans specific to the Pascal programming language that may be applied to many programming tasks. In order to change to a different task, all the plans must be replaced. In Apropos and Talus, the reference functions for sub-programs that are common to several programs could be reused.

Our system aims to to distinguish between task-specific knowledge and knowledge that is specific to the programming language. In order to change to a different task, the task-specific knowledge is replaced for the new task but the language-specific knowledge need not be changed.

2.4.5 Teaching Broader Programming Issues

Some of the limitations in Proust's scope are revealed when we compare its performance on the Rainfall problem with its performance on the Bank problem, and consider the reasons for the difference.

The problem specification for the Rainfall problem is as follows:

> Noah needs to keep track of rainfall in the New Haven area in order to determine when to launch his ark. Write a Pascal program which will help him to do this. The program should prompt the user to input numbers from the terminal; each input stands for the amount of rainfall in New Haven for a day. Note: since rainfall cannot be negative, the program should reject negative input. Your program should compute the following statistics from this data:
>
> 1. the average rainfall per day;
> 2. the number of rainy days;
> 3. the number of valid inputs (excluding any invalid data that might have been read in);
> 4. the maximum amount of rain that fell on any one day.
>
> The program should read data until the user types 99999; this is a sentinel value signaling the end of input. Do not include the 99999 in the calculations. Assume that if the input value is nonnegative, and is not equal to 99999, then it is valid input data. (Johnson, 1990)

This problem is specified in detail. Error conditions are described. Only one data type is needed, i.e. real numbers.

The problem specification for the Bank problem is as follows:

> Write a Pascal program that processes three types of bank transactions: withdrawals, deposits and a special transaction that says no more transactions are to follow. Your program should start by asking the user to input his/her account id and his/her initial balance. Then your program should prompt the user to input:
>
> 1. the transaction type;
> 2. if it is an END-PROCESSING transaction the program should print out (a) the final balance of the user's account (b) the total number of transactions (c) the total number of each type of transaction and (d) the total amount of the service charges, and stop;
> 3. if it is a DEPOSIT or a WITHDRAWAL the program should ask for the amount of the transaction and then post it appropriately.
>
> Use a variable of type CHAR to encode the transaction types. To encourage saving, charge the user 20 cents per withdrawal, but nothing for a deposit. (Johnson, 1990)

Proust was able to analyse completely only 50% of the Bank problems compared to 81% of the Rainfall problems. The rate of detection of bugs in the two programs was similar, but the Bank problem led to many more false alarms about buggy code that was in fact correct. Reasons for Proust's lesser performance on the Bank problem are:

- The Bank problem has more goals than the Rainfall problem.
- The specification of the Bank problem is less complete. It contains many implicit goals, for instance how to deal with a negative bank balance.
- The specification for the Bank problem does not include the same kind of useful hints to the pattern matcher as the Rainfall problem. For instance, the Rainfall problem explicitly restricts the terminating condition to be the input 99999, whereas the Bank problem only restricts the transaction types to being characters. It does not specify which characters are expected.

(Johnson, 1990) suggests that in order to be dealt with properly by Proust, the Bank problem would need to be specified in more detail. Johnson proposes that decisions in the problem domain, such as what to do about a negative bank balance, should be included in the specification, and that decisions about data structures, such as the exact codes to be used for transaction types should also be specified. This would certainly make life easier for the recognition software, but it does not encourage students to learn to make these decisions for themselves. Decisions about data representations are certainly part of a skilled programmer's task. Strictly speaking a programmer might not expect to decide what to do about negative bank balances, since this would probably be part of the system's specification. However noticing missing parts of the specification is an important part of most real programming tasks, and considering how to deal with them is a necessary skill. We propose an alternative approach in which decisions about data representations are left to the student and the analysis is able to deal with them.

CHIRON is a further development of Proust which deals with some of Proust's limitations in representing and recognising bugs (Sack, 1990). Like the recognition systems based on the Plan Calculus which are discussed in Section 2.6.3,

CHIRON recognises bugs by abstracting away from the details of the code. Its plans each reflect different aspects of the program, including syntax and data flow. It extends this recognition approach into debugging by using an abstract hierarchy of bugs. CHIRON is specifically intended to deal with high frequency bugs in novice students' Pascal programs: that is, with errors in syntax, typographical errors, incorrect ordering of operations and incorrect composition of the solutions to sub-problems. It does not deal with the other design problems that students face.

The tutorial debuggers that we have described in Section 2.4 reconstruct the student's programming decisions from the completed program and identify and correct bugs in complete programs. For instance, even though Proust's approach is analysis-by-synthesis, the only input to the process is the code of the completed Pascal program. In these systems the design decisions are reconstructed from the code itself. This has the advantage that a complete program offers a lot of context. If the program is large, then the repetition of similar bugs within the program makes it possible to infer the existence of underlying misconceptions (Johnson, 1990).

2.4.6 Limitations Compared to Tutorial Environments

The tutorial debuggers that we have described in this section are expected to perform their analysis on the student's best attempt at a complete program. The input program may or may not be required to be syntactically correct, but for these systems to operate, most of the program must be present. There are two problems with this "after the fact" approach to teaching programming.

The first problem is, how to gather information about the student's intentions in all aspects of the programming process. A complete program is only the end result of the programming process. It is not a complete trace of how the students performed the design nor of the difficulties they encountered on the way. The students may have encountered and corrected difficulties in the program design

which do not appear in the final program, difficulties which may have slowed down their programming and offered opportunities for teaching.

A closely related question is when to intervene with tutorial help. Tutorial debuggers deal only with complete programs and usually with syntactically correct programs. Novices may have difficulty in creating complete or syntactically correct programs in the first place. By the time students have completed the program they have corrected many bugs. A tutor which works on syntactically correct programs requires that students have already corrected syntax errors. Some approaches to solving a problem tend to lead to incomplete or incorrect solutions whereas other approaches are more likely to lead to correct solutions (Spohrer *et al*, 1985). Early intervention could save the student from considerable confusion and floundering.

The following section on tutorial environments describes an alternative approach to these problems.

2.5 Tutorial Environments for Novice Programmers

Human tutors can provide continual monitoring and structuring of students' problem-solving. Tutors diagnose students' confusions and provide appropriate instruction, they can communicate the overall structure of the problem and manage a student's errors. They supply immediate error feedback, put students on the right track and prevent students getting lost. Students with tutors don't give up so easily and learn from their attempts (Anderson *et al*, 1985). Tutorial environments reconstruct in an automated system the close monitoring and immediate feedback of a human tutor.

The LISP Tutor accepts code character by character as the student types in the program and intervenes as soon as it spots an error. Bridge

(Bonar & Cunningham, 1988) is also highly interactive, and it helps the student to specify the algorithms behind the code as well as the code itself.

2.5.1 Immediate Feedback in the LISP Tutor

The LISP Tutor monitors the student's problem solving and provides tutoring as soon as the student goes off the path (Anderson & Skwarecki, 1986). It is highly reactive and responds symbol by symbol. It helps with problem-solving and planning, offers exercises and leading questions at appropriate points, and has a menu interface to let students choose options that are not part of the program itself. It has an interface to deal with syntax details such as bracket matching.

The LISP Tutor is based on a production rule model of student programming, in which each production in the system corresponds to a meaningful cognitive step. It uses a *model tracing* approach, in which the system follows the student's cognitive steps and if any step is not meaningful, comments on it (Anderson *et al*, 1985). The system has both correct and buggy rules, with natural language templates attached to rules to provide error feedback.

Production rules can embellish a problem specification, can write or change LISP code, or they can set new subgoals. These goals need not be directly programming ones: they may be more general goals such as "look for a similar case and copy it" (Anderson *et al*, 1984).

The LISP Tutor has been successfully used for much of an introductory LISP course in which students teach themselves. The content of the course uses small LISP procedures each of which embodies a small number of specific techniques (Anderson *et al*, 1990).

2.5.2 Capturing Layered Plans in Bridge

The Bridge system (Bonar & Cunningham, 1988) attacks the problem of obtaining plan information from the student by explicitly guiding the student through

the process of planning the code, as well as through writing it. Bridge also takes a model tracing approach, but it divides the programming process into three phases, and guides the student through each phase. First, the student builds a set of step–by–step instructions using English phrases chosen from a menu. Next, the student matches those phrases to programming plans. Finally, the student matches plans to constructs in the programming language. In each phase, there may be many levels of detail. Each phase must be completed at all levels of detail before the next phase is entered.

Within the framework of a system which closely monitors and constrains students' programming behaviour, Bridge solves the computational problem of deciding which plans the student is following. Having accurate information about the student's intended plans, it can offer more support to the student building the program. It also cuts down the search space of programs that can be generated from incorrect plans by allowing the student to implement only from correct plans. Errors are not permitted to cascade from one phase to another.

2.5.3 Scope and Limitations

The model tracing method presents some difficulties (Anderson *et al*, 1990). When several rules could produce the same output, and especially when those rules are to do with high–level planning and they do not immediately produce any output, it is difficult to build up a correct and unambiguous picture of a student's intentions and misconceptions. This makes it difficult to offer the right tutorial advice or suggest the best correction.

Requiring that students explicitly tell the system when they have made such decisions places a heavy burden on the students and may interfere with their performance of the programming task.

The LISP Tutor imposes very severe constraints on the way that students can build their programs. (Anderson *et al*, 1984) found empirically that novice LISP programmers tend to program top–down and depth–first, and this pro-

gramming strategy is built into the LISP Tutor. However, other studies e.g. (Rist, 1991) have found that novices do not follow such a strict strategy and tend to focus on particular parts of the task first, and forcing students to program top-down and depth-first imposes too strong a constraint on them (Anderson & Skwarecki, 1986). More recent systems based on the LISP Tutor (e.g. GIL (Reiser *et al*, 1989)) do not impose such a strict strategy. GIL offers a more flexible strategy using the dataflow of the program, a strategy which allows the student to build code either forward from the inputs to the program or backward from its outputs.

The LISP Tutor and Bridge are examples of successful tutoring systems that use analysis-by-synthesis and immediate feedback (Anderson *et al*, 1990; Bonar & Cunningham, 1988).

Immediate feedback has many benefits, but it also has problems (Anderson *et al*, 1990). Given a little more time, students might correct errors themselves. By seeing the effects of their errors students learn about the problem domain and by correcting their own errors they learn error-correction strategies within that domain (Foss, 1987). As students improve they tend to dislike immediate correction. When an error is detected there still may not be enough context to explain the error or to understand the misconception that underlies it. (Anderson *et al*, 1990) suggest waiting at least until the student has completed a line of code rather than one statement. The ideal time to present tutorial interaction is still to be determined.

In our work, we focus on tutorial debugging. Our objectives are to allow slightly more advanced students the freedom to make and correct their own mistakes, and also to explore the extent to which it is feasible to present large segments of code for analysis.

2.6 Support for "Real" Programmers

Students must eventually learn to deal with all the issues in programming. As students move on from limited exercises that emphasise particular aspects of the programming language, their programming exercises involve many aspects of design: interpreting problem specifications, breaking problems down into their constituent parts, choosing data representations and choosing algorithms. If we are to tutor intermediate students about how to program we must be able to recognise design decisions and critique them.

Exercises for beginners focus on canonical examples of a few aspects of the language. The exercises typically include detailed specifications which are given in terms of the programming language and programming concepts. Students are likely to try to solve the right problem and the choices of solution strategy are very limited. At this level, Proust has been quite successful as a means of detecting and critiquing a range of semantic errors in Pascal programs (Johnson & Soloway, 1984) and the LISP Tutor has been successful in encouraging students to learn the semantics of LISP (Corbett *et al*, 1993).

When we consider how to tutor more advanced students automatically, the problems become much harder. The specification may also be incomplete, leaving the student to decide exactly what behaviour is required from their program. The space of possible designs for these programs is large. Program specifications are given in terms of a problem domain, which leaves the student with many decisions about how to map from the problem domain into a programming language (Kant, 1985).

Existing systems for tutoring software concentrate on teaching, and hence understanding, the aspects of the design that relate closely to the programming language that is being taught. The majority of these systems do not deal with the problem of creating abstract solutions to problems or connecting the requirements of the task in the problem domain with the solution to the problem.

Some systems that support more experienced programmers in writing production software have focussed on these issues. We will now consider whether the approaches used to support more experienced programmers could be applied to similar problems faced by novices.

Common objectives in this field are the use of standard software components to support the design of new software (Rich & Waters, 1990) and the understanding, redesign, and reuse of existing software (Wills, 1990; Biggerstaff, 1989; Letovsky, 1988; Waters, 1988). Libraries of components are created at different levels of abstraction, which can be used to design new code as well to understand, reuse and translate existing code.

2.6.1 Cliché–Based Design

The Programmer's Apprentice project is a long-term research project whose aim is to use AI in a design support system for software development (Rich & Waters, 1990). This project aims to build up reusable libraries of commonly occurring program structures called clichés. It is intended to support the development of large programs. Programmers writing new software can assemble clichés to fulfill the necessary tasks. The project also aims to create an intelligent assistant (the Apprentice) which uses information in the library to perform routine subsidiary tasks when the software engineer has made higher level decisions. The programmer and the Programmer's Apprentice share the library of clichés. It is intended that eventually the Programmers Apprentice will support implementation, design and requirements capture.

The Programmer's Apprentice project has built KBEmacs, a tool which supports the implementation of LISP and Ada programs (Rich & Waters, 1983; Rich & Waters, 1990). Programs in KBEmacs are represented in an abstract language, Plan Calculus, in which different levels of abstraction are represented by plans at different levels. Clichés are represented by a hierarchy of plans. Each cliché includes information about which parts of the cliché may be varied, which must remain the same and which are defaults. Programs can be written by

naming the required cliché and specifying the variable parts. KBEmacs assists the programmer by doing mundane tasks such as filling in variable declarations, and it can propagate changes through a program with little effort from the programmer. So the Implementation Apprentice enables a programmer to write a relatively short specification and generate a considerable amount of code from it. It is now being extended to form a Design Apprentice which will include design clichés (Rich & Waters, 1990). The Design Apprentice will automatically make sensible choices about how to implement the design (which the programmer may override) and it will be able to detect and explain errors made by the programmer (such as omissions which lead to the Apprentice being unable to implement the design, and inconsistencies between the programmer's requests and the selected clichés).

Clichés abstract up from the programming language. For example, a cliché that manipulates lists can manipulate any data structure that has a head and a tail. A looping cliché can be implemented using a loop statement, tail recursion or a goto. The representation of a cliché in Plan Calculus abstracts away from both the programming language and also away from specific implementations in a given programming language. A cliché expressed in Plan Calculus can be manipulated more easily than code.

Plan Calculus expresses the operations of the program and the data flow and control flow between those operations. Language primitives are divided into those which operate on data structures (e.g. LISP `car`) and those which deal only with control flow (e.g. LISP `cond`). Plans may be drawn as directed graphs in which operators are nodes and the control and data flow are connections between them. Clichés are organised into a hierarchy, so that one cliché may be used as part of another larger cliché.

The Plan Calculus formalises the relationship between clichés using overlays. An overlay consists of the plans for each of two clichés and the correspondence between them, in particular the idea that one cliché implements another

(Wills, 1990). Then one plan can be replaced by another, to give a more efficient implementation or a more abstract description of the cliché.

2.6.2 Design Recovery

Clichés offers a language in which to describe some design decisions (i.e. the choice of clichés for specific problems) and KBEmacs offers the potential to record these decisions and associate them with the resulting the code (although in fact it does not record them (Wills, 1992)). Recording design decisions certainly simplifies recognition and it provides context for critiquing. However, it cannot make the problem of reconstructing the design decisions from the code go away altogether. Some parts of the design system impose constraints on the program that it must fulfill, and the system would be expected to check or enforce those constraints. Other parts of the design act as documentation for a human reader but they are not enforced by the system upon the program itself, in which case the program may not actually correspond to the design specification.

If the automated system is to check the code against constraints, it is easier to check a program against a specification than it is to reconstruct a specification from the program. Such a check can identify a problem and localise it. But when it comes to describing exactly what has gone wrong in the program and proposing a good repair, then some reconstruction of what the code actually does becomes necessary.

If the software has got out of step with its design specification then it is important to reconstruct design decisions from the code itself, to compare them with the documented decisions and to decide whether it is the code or the design documentation that should change.

Cliché–Based Design Recovery

The Programmer's Apprentice project combines both software design and the reconstruction of design decisions in existing programs, using the same represen-

tational formalism (clichés and Plan Calculus) for both (Rich & Waters, 1990). GRASPR uses the Plan Calculus to recover design decisions from LISP programs (Wills, 1990; Wills, 1992). It uses clichés to deal with syntactic variations (e.g. different ways to implement a loop in LISP), with variations in algorithms which must at some higher level of abstraction be recognised as implementing the same concept (e.g. different ways to create and operate on a hash table), with overlapping algorithms which may arise from optimisation, with unrecognisable code (that may not arise from a cliché), and with clichés that are scattered through the code.

GRASPR uses overlays to handle descriptions of abstract data structures. The overlays enable the parser to produce a hierarchical description of the program. The end result is a derivation tree for the program which captures the relationship between different syntactic versions of the same abstract cliché.

GRASPR first analyses the program to create a flow graph representation of its control and data flow. It then uses a graph parsing approach to parse the program into its constituent clichés. It deals with merged plans and optimised code by allowing the same bit of code to form part of more than one cliché. In effect, it undoes the optimisation.

GRASPR deals with the problem of partially understanding code which contains some buggy code and some correctly implemented code. Its parser can create a forest of derivation trees rather than needing to create a complete tree, so that partial recognition is possible. It starts parsing at each point in turn in the program graph and can finish parsing before it reaches the end of the code. Parsing can start at any level of abstraction, not just at the most abstract level, so a low-level cliché may be recognised even if it isn't used in any expected way as part of a high-level cliché. GRASPR does not deal with the problem of understanding the buggy sections of code. It simply skips buggy code, and it does not make any hypotheses about the purpose of buggy code based on the context in which it is found.

Plan–Based Design Recovery

The goal of CPU (Letovsky, 1988) is to understand and document programs by reconstructing the design. Letovsky uses programming plans to create abstract descriptions of programs. The CPU system takes Fortran programs, converts them them to a lambda calculus notation and then picks out particular structural patterns which can be replaced by summaries. Different plans are written in terms of the lambda calculus and a small set of stream operators. Whenever the body of a plan is recognised in the transformed program, the plan can be removed and replaced by the goal of the plan. The approach is similar to cliché–based design recovery but CPU describes programs in a plan–based language that is more specific to a particular problem domain.

Recovering Informal Aspects of Design

Cliché and plan–based approaches to design both capture some of the more formal aspects of a program and its design, which can therefore be represented formally. Desire is a system that focuses on the automated understanding of the structure and concepts in a program, including informal concepts (Biggerstaff, 1989). Desire combines reconstruction with recording. Reconstruction of the code is performed using libraries of structures (similar to clichés) that are matched to the code. The software re–engineer can record high–level abstractions as well as have them inferred automatically from the code.

Biggerstaff's model of design recovery records informal information such as variable names and comments which are linked to form a model of the problem domain. The domain model includes knowledge about program structures and language structures, and it also contains knowledge about problem domain structures and naming conventions which have only been known to expert software engineers and application domain specialists. Desire distinguishes between concepts in the domain and the entities that represent those concepts. The domain model combines a semantic net representation with a hypertext system.

2.6.3 Scope and Limitations

Systems which tutor novices start with some major advantages over systems which support experienced programmers. Systems for novices can deal with problems whose solutions are known, whereas experienced programmers are trying to write programs for novel situations. In fact, not only does a teaching system for novices know the correct algorithms, it may also know the most common mistakes. Expert programmers, on the other hand, are usually trying to solve novel problems. At best, their programs will be variations on known themes.

CPU has been demonstrated on programs with tens of lines of code and also with one 300 line FORTRAN program which manipulates a personnel database. GRASPR has been tested on two Common LISP programs which simulate parallel message-passing and are approximately 800 lines long. The programs were written for use, not as exercises. Both programs have been studied intensively to derive the necessary clichés. This is some way away from the ideal for design reconstruction, which is to reconstruct production programs of tens of thousands of lines of code. Nevertheless this gives some hope of success with the 100–250 line student programs we describe in Chapter 3.

Variations in implementations remain a problem for these systems. Before GRASPR can operate, some transformations are done to the programs by hand (Wills, 1992). The user must also tell GRASPR where to locate some clichés and to skip kinds of code which cannot be recognised (e.g. multiple recursion). GRASPR does not recognise bugs, but skips sections of code that are unrecognisable. An initial attempt has been made to apply Plan Calculus to debugging Pascal programs (Lutz, 1991). Initial results seem promising, but this analysis has been applied only to domain-independent Pascal clichés and to small programming exercises such as sorting and some aspects of the Rainfall problem (Lutz, 1993). The method of summary used by CPU is effective for correct programs, but it has not been applied to buggy programs. CPU can only recognise a plan when all the components of a plan have been recognised. A fully automated analysis of buggy programs several hundred lines long has not yet been achieved.

2.7 Conclusions

Novice programmers need to learn a wide range of skills. Automated systems can help novices to learn those skills. We have described the systems that exist to provide automated support to novice programmers. These systems deal successfully with the small and well-specified programming exercises that novices use early on, but we find that they are limited in dealing with the complete range of decisions and the complete range of errors that students may make, especially as novices become more experienced and move towards larger programs with less detailed problem specifications. We have found that the automated support that has been provided for program construction by experts could be used to support novices.

We are interested in the problems faced by students who are learning to write programs that are not specified in detail, that require the student to make many different kinds of programming decision, and that require the student to implement many sub-tasks. Existing automated tutoring systems for computer programming have been successful at providing support for novices' problems in learning about the syntactic and semantic aspects of programming languages. They deal with some of the problems but many remain unsolved and even unexplored.

Different tutoring systems have traded off various difficult aspects. Tutorial debugging approaches which use knowledge about the task deal with a limited number of small and thoroughly specified programs. They focus on correcting bugs which make programs behave incorrectly, rather than on correcting bodges which only lead to poor design. When the student's intended idioms and algorithms are recognised, these debuggers use this information to provide context for identifying bugs in the implementation. They do not critique the student's choice of algorithm, good or bad.

Some of the abstraction techniques that have been devised to support the construction of professional software might also be applied to the understanding of novice software. To do this entails extending their recognition mechanisms to deal with badly designed code. It requires the representation of design decisions in the problem domain as well as decisions that are specific to the programming language.

An architecture for understanding novices' design decisions must represent decisions in both the domain of the problem being solved and also the programming language. The system must be able to separate these decisions, in order to support tutoring and also to allow the system to be easily adapted for a variety of programming exercises.

Chapter 3

Motivation: Design of Intermediate Prolog Programs

3.1 Introduction

In this chapter we describe a study of completed Prolog programs written by intermediate student programmers. The objectives of the study are:

- to understand intermediate student's errors in terms of a model of the software design process;
- to compare the errors made by intermediate students with the models that have been used to tutor novice programmers;
- to investigate the suitability and limitations of such models for intermediate programmers;
- to identify design decisions which could feasibly be detected and critiqued in a tutoring system for intermediate programmers.

We present a model of software design skills and relate the design errors in the student's programs to this model. We present an analysis of students' errors in terms of this model. We compare the errors that we found to errors typically made by novice programmers. The errors that we found can be attributed to the students' inexperience in Prolog programming and to their inexperience in

general software design. A few errors were due to students' lack of knowledge about the problem domain, the specific game-playing strategies that they were required to implement.

We find that the models of errors that apply to novices working on very constrained tasks account for only a few of the errors found in these students' programs. Intermediate students understand the Prolog programming language better than novices, and so their programs contain fewer errors that relate to Prolog's logical or execution structure than those written by novices. However, many intermediate students still have apparent difficulty in creating well-designed Prolog programs. We explain this difficulty in terms of the skills that are required in the software design process.

We describe the criteria we use for critiquing software and the problems entailed in automatically detecting and critiquing design errors.

We conclude that an automated system to detect and critique design decisions would be useful in teaching software design. In particular, a system which could detect and critique particular decisions about data representations and programming techniques would be a useful and interesting development of existing tutoring systems for the teaching of computer programming.

3.2 The Study

We studied nine completed Prolog programs written by students as assessed exercises in their seventh week of a nine week Prolog course. The programs played a game of noughts and crosses between the computer and a human opponent. The programs typically consisted of between 150 and 300 lines of code. All the programs we studied ran and played the game successfully. Nevertheless, all of the programs contained some important design flaws.

3.2.1 The Problem Specification

The task statement required that the program maintain, display and control the state of the game, as well as accepting the user's input and playing correctly itself. The level of skill that the computer was to have was not specified. All the students were required to use a common top-level framework for their code. The top two predicates were specified as being generic for two-player games and were taken from Sterling and Shapiro (Sterling & Shapiro, 1986). These predicates called half a dozen other named predicates which the students supplied. The variables manipulated by the predicates were required to serve particular purposes. However, the tasks were loosely specified and the exact contents of variables were left to the students.

We now quote the problem specification as it was given to the students:

MSc Prolog Tutorial Exercise 7 ASSESSABLE You are given the following framework for a game playing program, taken from Sterling & Shapiro *The Art of Prolog* pages 296–297 :

```
play(Result):-                  % To play, with some final Result:
   initialise(State, Player),    % First make an initial game state
   display_game(State, Player), % Then display the game
   play(State, Player, Result). % Then play the game, with some Result

play(State, _, Result):-
```

```
   end_state(State, Result), !, % If some terminating state is reached
   announce(Result).            % Then just report the result
play(State, Player, Result):-   % Otherwise, the current player
   choose_move(State, Player, Move),          % chooses a move.
   move(Move, State, State1),   % That move is implemented
   display_game(State1, Player),% The game is redisplayed
   next_player(Player, Player1), !, % We find the next player
   play(State1, Player1, Result).   % And give him/her/it a turn
```

Copy this section of code from the file and use it as the top-level control mechanism for your program.

Your task is to add definitions for `initialise/2`, `display_game/2`, `end_state/2`, `announce/1`, `choose_move/3`, `move/3` and `next_player/2` which, in combination with the code given above, is a program for playing an interactive game of "noughts and crosses" (sometimes called "tic-tac-toe") with the computer.

What is "noughts and crosses" ? This is a simple board game played by two persons (or computers). The board consists of a grid consisting of 3 rows and 3 columns, i.e. :

```
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
```

Each player takes a turn to put a mark in a single square - usually one player uses a cross (x) and the other a nought (o), hence the name. The winner is the first person to get a horizontal, vertical or diagonal straight line of 3 of their marks. For example, these are winning states for whoever has the "cross" mark.

```
-------------   -------------   -------------
| x | o |   |   | x |   | o |   | o |   | o |
-------------   -------------   -------------
| x |   | o |   |   | x | o |   | x | x | x |
-------------   -------------   -------------
| x |   | o |   |   | o | x |   |   |   | o |
-------------   -------------   -------------
```

Hints `initialise(State, Player)` sets up the initial State of the game and decides the Player who plays first. You must decide what is a good representation for a board state (list ?, substructure ?).

`end_state(State, Player)` detects that State is some terminating state of play. This could be either when all the squares contain marks but nobody has won (a draw) or when some player has won.

`display_game(State, Player)` should print out a description of the current state of play. You can make this as complex as you like - from just writing out State to printing a diagram of the board. If you want to clear the screen between displays you might try using the goal `shell("clear")`.

`choose_move(State, Player, Move)` is the predicate for choosing a valid Move given the current State and Player. If the Player is the 'user' then this should prompt him/her for a choice of move and check that their choice is a valid move, given the current State. If the Player is the computer then it should automatically generate a reasonable move.

Now you might immediately think of implementing some sort of game tree search mechanism to perform this task but you DON'T have to get that fancy. However, you will get some credit for implementing a simple mechanism which does more then just select the first unoccupied board square. For example, the program might first look for a place where its opponent could complete a winning line of squares and block that move.

`move(Move, State, State1)` just implements the Move obtained from `choose_move/3`, returning the updated state, State1.

`next_player(Player, Player1)` is responsible for swapping players between stages of the game. This should be dead easy.

Format of practical submissions To make the job of marking easier and to avoid misinterpretation of the decisions you have made when writing the code, I would like your answer to contain the following information:

- The code (of course) well documented.
- Succinct explanations of why you chose a particular representation of the board state and particular strategies for the automatic move generation program in `choose_move/3`. You can mix this in with your code documentation if you wish.
- A transcript of a sample run of your program. A handy way to get a transcript of your Prolog session is to use the Unix command `script`. See Unix manual for details.

3.2.2 The Problem Domain: A Game of Noughts and Crosses

The problem specification briefly describes the game of noughts and crosses. The winner of the game is the first player who creates a line of three of their own tokens, in a row, column or diagonal.

A good player can always force a draw, no matter which player starts. This level of play was not part of the problem specification, and some students claimed they had deliberately created players that played a less than ideal game so as to let the human player win occasionally.

Certain concepts in the game domain are essential for solutions. Some are mentioned in the problem specification, such as lines, squares and marks, and special cases of these such as rows, columns and diagonals, occupied and unoccupied squares, and noughts and crosses. Other concepts are mentioned though not named. For example, the problem specification describes a potential line:

> ... the program might first look for a place where its opponent could complete a winning line of squares and block that move.

The student must interpret this to mean a line which has two of the opponent's tokens in place and a third unoccupied (empty) square.

Other game-playing concepts are created and used by the students as they develop their programs.

A distinction is needed between different kinds of squares, that is, between the corner squares, the centre square, and the side squares (the remainder). Each of these kinds of square is involved a different number of possible lines - each corner square is part of three lines, the centre square is part of four lines, and each side square is part of two. There is a symmetry in that all squares of the same kinds have similar properties. The centre square or one of the corners makes a good opening move; it does not matter which of the corners is chosen.

3.2.3 Discussion

Exercises for beginners focus on canonical examples of a few aspects of the language. The exercises typically include detailed specifications which are given in terms of the programming language and programming concepts. Students are likely to try to solve the right problem and the choices of solution strategy are very limited. At this level, Proust has been quite successful as a means of detecting and critiquing a range of semantic errors in Pascal programs (Johnson & Soloway, 1984).

This exercise reflects the acquisition of a broader range of programming and program design skills. Compared to the exercises dealt with in other intelligent tutoring systems for computer programming, it comes closer to a real-life programming problem in the following ways:

Incomplete specification The functional specification for the exercise is incomplete (Simon, 1973; Visser & Hoc, 1990). The students must translate a specification that is given in terms of the problem domain into a computer program that can be run. They must devise data representations, rather than only design operations on data representations that have been specified.

Scale The students must create and combine larger components and deal with the additional memory load that this implies, especially when considering the interactions between components (Adelson *et al*, 1985). There are a vast number of design decisions to be made. The impact of a decision made in one part of the code on other parts may not be immediately obvious.

Multiple solutions There is no single "best" solution to the programming problem, nor even a small and easily enumerated set of solutions. The player may be implemented using many different algorithms, although some algorithms may be preferred as producing a better player or better design (Visser & Hoc, 1990).

Design evaluation Students must begin to judge the quality of the design that they create and evaluate their own program against design criteria (Kant, 1985).

3.2.4 The Students

The students had a wide range of backgrounds. They were all studying for an MSc conversion course in Artificial Intelligence. Many of the students had significant computing experience but few had formal computer science qualifications, and few had previous experience in Prolog. All had learnt LISP for a term before the Prolog course, and the LISP course taught some elements of software design. For some, the MSc was their first experience of computer programming whereas others had considerable previous experience programming in procedural programming languages. This was reflected in a wide range of programming styles and program design skills.

The students are referred to by identifiers P1–P9 in the discussion. These identifiers are also used in the catalogue of program design errors in Appendix A and the examples of their code for finding an empty square in Appendix D.

3.2.5 The Study Set

Out of the thirty students' programs, two were rejected immediately because they did not run. From the remainder, nine programs were chosen at random for analysis. All of the programs fulfilled the functional specification in that they contained no syntax errors and they played a legal game with the user. In the study set we found examples of many kinds of design decisions and many kinds of errors in those decisions.

We now present a model of the software design process which can be used to explain many of the design errors that we found. Then we suggest some broad classifications for where the errors that we found arose.

3.3 The Software Design Process

The design process has been characterised as one of solving ill-structured problems (Simon, 1973). Brown and Chandrasekaran have identified three classes of design problem, with varying amounts of structure (Brown & Chandrasekaran, 1989). Class 1 is open-ended creative design in which the goals are not clearly specified and there are no ready-made ways to decompose the problem into smaller elements. Class 2 is design for which the problem can be decomposed in standard ways but major components of the design must be created from scratch. Class 3 design is relatively routine. Effective problem decompositions are known and there are standard plans for solving the component problems. Class 3 design is not trivial, because choosing the right plans is a complex problem in itself.

One model of the design process that has been used in artificial intelligence is to treat design as a search process through a set of states with the objective of locating a goal state. This approach is applicable to Class 3 design. In software design, the goal state is specified by how the designed artefact will behave, i.e. the functional specification for the program. The current state is not specified as behaviour but by the mechanisms that will eventually support that behaviour, i.e. the program (Adelson & Soloway, 1985). The degree of structure that can be imposed on this search problem varies according to how familiar the software designer is with the domain in which the design is being created, how similar the design problem is to ones that have been solved before, and how familiar the designer is with general design principles.

The situation of novice programmers is in some ways similar to the situation of more experienced software designers faced with an unfamiliar problem. Novices are uncertain as to how to break the problem down and they do not have a large collection of pre-formed plans for solving parts of the problem (Rist, 1991). Even problems that would seem like a very straightforward Class 3 problem to

an experienced software designer might seem more like a Class 2 or even Class 1 problem to a novice. We would expect to find novices using idiosyncratic solutions rather than standard ones, or misusing standard solutions.

Adelson and Soloway observed experienced software designers who were building a system, and hypothesised that designers form a sequence of mental models of the system being designed so far. Each model in the sequence is a top-down breadth-first refinement of the previous one, and each model can be mentally simulated to predict the behaviour of the system (Adelson & Soloway, 1985). A mental simulation allows the designer to check that the mechanism really will exhibit the desired behaviour at each level, before progressing to the next level.

(Adelson *et al*, 1985) observed experienced software designers working in a problem domain with which they were familiar, and hypothesised that these designers work in the following way:

- Before forming any mental models, designers make assertions about how the system will behave and they explore the design implications of these assertions. As a result, they constrain the range of possible designs. These assertions make the design concrete enough to run.

- Each mental model is an expansion or refinement of the last. The sequence of models can be seen as a tree being expanded in breadth and depth, progressing from the abstract to the concrete. One level of abstraction is handled at a time and each level of abstraction is only a little more detailed than the last, i.e. a form of breadth-first expansion of the tree.

- Designers repeatedly run mental simulations on these mental models. One important use for these simulations is to check for unforeseen interactions between components. Simulations can only be run at a single level of abstraction, which means that the breadth-first development of the mental model is critical to simulation.

- When the designer is working at one level of abstraction and notices a problem or an opportunity at another level of abstraction, a mental or written note is made of it for future reference. The note is dealt with when the designer is ready to deal with that level of abstraction. This enables the designer to work at a consistent level of abstraction without losing track of important ideas as they arise.

- If the designer already knows how to solve a sub-problem then this complete solution plan is retrieved by name.

Designers also evaluate their designs against explicit design criteria (Kant, 1985; Joni & Soloway, 1986). Designs are evaluated at each stage of development, not only at the end. Design criteria may vary and they may even conflict, for example run-time efficiency and readability.

Recent research suggests that even experienced software designers working in a familiar domain do not always proceed in a purely top-down breadth-first manner but mix this strategy with opportunistic expansions of the design (Visser & Hoc, 1990). They may start in the middle of the design tree or digress to other levels of the design breaking off from an incompleted high-level design to design some low-level primitives. They are also found to modify earlier design decisions even if these occur at higher levels than the current one. Software designers still create top-down decompositions of the problem, but their predicates for creating the decomposition are opportunistic and non-monotonic.

As a result, we would not wish to go as far as tutoring systems such as Bridge (Bonar & Cunningham, 1988) which impose a top-down design strategy. We are exploring what can be done by allowing the student to complete a program by whatever method, and then critiquing the design decisions they have made as their results appear in that program. We can usefully present a critique in terms of a top-down model of design at different levels of abstraction, even if the design was not developed in this way.

3.4 Novice Behaviour In Program Design

Software design requires the exploration of several different problem spaces (Kant, 1985). Kant observed that to devise an algorithm independently of any machine architecture or programming language, designers must be able to explore two problem spaces. The first is a space of algorithm design which describes what is achievable in standard computer systems and standard design methods. The second is that of the application domain space, in our example that of game playing strategies. We are looking at implemented programs in Prolog and so students are also exploring a third problem space, the space of Prolog program designs (Rist, 1991). These three problem spaces for programming are summarised in Figure 3-1.

We find that design errors arise in exploring each of these domains: the Prolog domain, the general software design domain and the AI and game playing domain. We therefore discuss the design errors that we found in each of these problem spaces. Detailed examples are given in Appendix A.

3.4.1 General Software Design

Studies of novice software designers show that novices differ considerably from experts in the way that they design programs. Novices do seem to create mental models of the design but typically these models cannot be run as simulations. As a result, novices cannot check the consequences of interactions between design decisions in different parts of the program (Adelson *et al*, 1985; Adelson & Soloway, 1985).

Novices typically go straight from the specification to a very concrete level of design (pseudocode (Adelson *et al*, 1985) or code (Rist, 1991)). They do not build mental models at different levels of abstraction. They develop their design depth-first rather than breadth-first, concentrating on easy aspects of the design

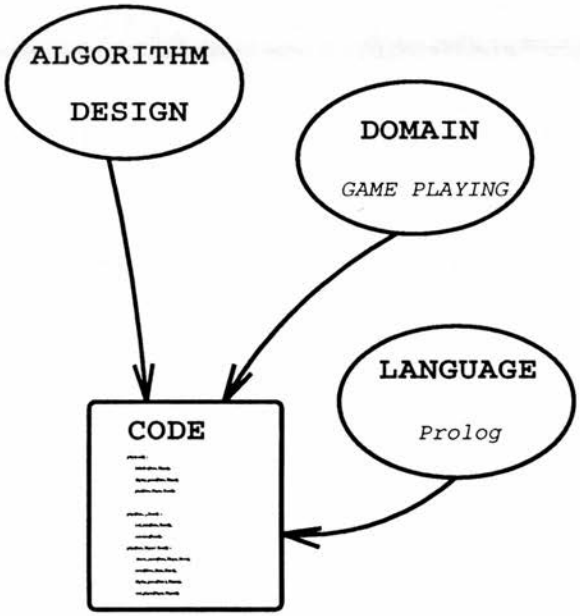

Figure 3–1: Design spaces for Prolog programming

first. They do not make notes for future reference but deal with issues as they arise (Adelson & Soloway, 1985).

Novices are less able than experts to check their programs against design criteria. They may lack the criteria (Joni & Soloway, 1986), or else their failure to design a complete model stage by stage means that design criteria can only be applied locally until the program is complete (Kant, 1985).

Rist found similar differences in programming behaviour between novice and intermediate programmers to those that Adelson found in algorithm design behaviour among designers with different degrees of experience (Rist, 1991). Intermediate programmers are able to retrieve partial solutions for a task and merely have to fill in the details, whereas novices must construct their solutions from scratch. As the problem becomes less familiar, so more experienced program-

mers behave more like novices, creating new solutions instead of retrieving partial solutions.

3.4.2 Design in an Unfamiliar Problem Domain

The degree of familiarity with the problem domain makes a difference to the methods that designers use and their success. Adelson and Soloway have studied designers working in a familiar domain (Adelson & Soloway, 1985). They found designers who are re-designing a familiar object or component simply retrieve an existing mechanism for that object or component. They do not simulate the behaviour of familiar parts of the model, and they only simulate the behaviour of parts that they have not designed before. Designers constructing an unfamiliar object but still in a familiar problem domain, construct models and run mental simulations on them. They use their familiarity with the domain to make assumptions about the behaviour of the system which constrain the design and to explore the likely consequences of those assumptions. They are able to simulate the whole system and were aware of interactions between parts of the system.

Kant has studied experienced software developers designing an algorithm in an unfamiliar problem domain (Kant, 1985). When neither the domain nor the object are familiar, then domain experience cannot be relied on and designers must rely on general knowledge and skills. Kant found that experienced designers working in a totally unfamiliar domain also constructed and ran models. But they left out some important assumptions, and they were unable to simulate the interactions between different parts of the system. Designers use more opportunistic and less monotonic design strategies to deal with unfamiliar problem domains.

3.4.3 Design in an Unfamiliar Programming Language

Novices learning a new programming language make many mistakes because they have a flawed understanding of the behaviour and structures of primitives and the

Prolog execution model (Fung *et al*, 1990). They are also ignorant of the standard programming techniques in the programming language (Brna *et al*, 1991). This is true of novices in Prolog even if they are experienced in general software design and in other programming languages (White, 1988).

3.5 Design Criteria

There are a number of different design criteria by which we can assess a program. These criteria may be behavioural, e.g. that the program must have a particular functionality, or that it must produce output of a particular form, or that it must run efficiently on large quantities of data. They may also be structural, e.g. that the code should use standard and recognised programming techniques.

In the programming exercise we have studied, the single most important design criterion is that the program should fulfill its functional specification. This meant that the program should behave properly, accepting moves from the user, responding with legal moves of its own and displaying the state of the game on each turn. We studied only programs that fulfill this design criterion. The functional specification is incomplete, and so other behavioural issues arise to do with the effectiveness of the player.

Some criteria are specific to particular design or programming languages. In Prolog, a declarative programming style is often valued which makes only minimal use of the instructions for explicit control of execution. Other criteria may be specific to the problem domain. For example, unnecessary search must be avoided in a large search problem, whereas if little search is required then duplication of search or the following of unnecessary paths is acceptable.

There are bound to be trade-offs between design criteria. A fast algorithm may take up more memory than a slow one. Code that is written in a standard declarative style that is easy to read and to modify may be less efficient to run than code that is written in a specialised style using explicit control that is

difficult to read or modify (although see (O'Keefe, 1990) for an argument that a declarative Prolog program can be more efficient than using explicit control). We can give different weight to different design criteria for different problems in different context.

In our study of students' design errors, we have used the following criteria, without prioritising any one:

- use of standard and appropriate programming methods;
- correctness;
- robustness;
- clarity and readability;
- appropriate use of Prolog's declarative features.

These criteria are applied in more than one problem space. In the spaces of AI and game–playing, standard algorithms may be applied for heuristic game–playing or for deep search, and in the space of Prolog programming, standard programming techniques may be used for many detailed aspects of the implementation. Correctness applies to the behaviour of the game–playing program as a whole and also to individual predicates in that each predicate should fulfil its own specification. Robustness applies particularly to the Prolog space, in that individual predicates should work not only in the specific context of that one noughts and crosses program but they should also work correctly for all reasonable inputs, and not depend on the surrounding context to provide only the correct inputs.

Clarity and readability apply to all three problem spaces: the code should make explicit the concepts it used in the game-playing domain, it should exhibit clearly the structure of the chosen algorithms, and it should be possible to follow the the Prolog code structure and easily infer its execution behaviour. The appropriate use of Prolog applies only to the Prolog problem space.

Efficiency of execution is also considered, but we treat it as secondary to these other criteria.

3.6 An Analysis of Prolog Students' Design Errors

In our study of students' design errors, we found errors in all three problem spaces — those of algorithm design, those of AI and game-playing, and those of the programming in Prolog.

In the following subsections, we consider the errors that we found in the students' programs as they could be attributed to difficulties within the three problem spaces in which the students were working (Figure 3-1):

General design of algorithms In this exercise, our main criterion for general skill in algorithm design is how well-structured the program is. Other design criteria such as run-time efficiency are less important here.

Design of algorithms in the domain of AI game-playing
Noughts and crosses is a game in which very simple heuristic strategies can result in very effective players and complex search-based strategies are not essential, and so this problem required little skill in choosing game-playing strategies.

Design of Prolog programs Specific skill in designing programs in Prolog is reflected by the correct use of Prolog primitives and the use of recognisable and correctly implemented programming *techniques*.

We did not observe the students while they were writing the programs. Students were asked to provide comments describing their design decisions and the comments suggested explanations for some of the errors that arose. Nevertheless, we cannot be exactly certain of how all of the errors arose. The same surface errors as they appear in the code may be attributed to different problems, and even to problems in different design spaces. In spite of this difficulty, errors may usefully be explained in terms of one or more design spaces.

These three design spaces are a useful framework in which to analyse bugs. Each design space is a natural topic for teaching and the different design spaces provide the context in which to formulate comments to the student.

3.6.1 General Design of Algorithms

Novice designers do not know how to structure their problems properly. They have difficulty in mapping from the task structure in the problem to the structure of the solution, in distributing the sub-tasks among code (Johnson & Soloway, 1984).

Structural decisions break the problem down into appropriate components and determine which aspects of the problem are to be modeled as data structures and which as control. The two main design problems are the failure to make concepts and relations explicit and the failure to abstract and hide implementation details. Difficulties in creating an appropriate structure for a solution are further revealed by:

- Implementations at the wrong level of abstraction;
- Scattering a single task among different parts of the program;
- Repetition of code;
- Repetition of execution;
- Hidden bugs;
- Mis-use of data constructors;
- Mis-use of control (especially cuts).

In the following subsections, we discuss these difficulties in more detail.

There are also structural decisions to be made about which elements of the game are represented explicitly as variables, which are implicit in the computation and which are not included in the program. In this exercise the students were instructed to create particular variables at the top level for particular purposes, and so we found no examples of design difficulties here.

Levels of Abstraction

It is important that the structure of the program should reflect the structure of the tasks and the concepts within that program. Conceptually separate components of the program should be created as structurally independent components with well-defined interfaces to the rest of the program.

The right level of abstraction should be used both in the design of predicates and in the design of data structures that the program will create and manipulate.

For predicates, the structure of the program should reflect the structure of the tasks to be performed. As a rule, each Prolog predicate should perform a single task, with calls to other predicates to perform sub-tasks. It is easy for novices to fail to reflect the structure of the tasks and sub-tasks in the code that they write. Novices try to make a single predicate do too much work. For instance the task of detecting the end of the game consists of two distinct sub-tasks, those of detecting a win by either player and of detecting a draw. Most students therefore wrote the predicate `end_state/2` so as to call two separate predicates, one for each task. Some students only wrote a separate predicate for the more complex of the two tasks, detecting a win, leaving the very simple task of detecting a draw at the top level. P3 wrote only a single predicate to detect the end of the game. This made the structure of the code to detect the end of the game quite obscure.

Some wrote long linear sequences of pattern-matching clauses, in which the patterns in different clauses represented very different kinds of condition. These are better broken down into separate predicates for finding particular kinds of pattern. Structuring code in this way makes it easier to ensure that all the patterns of a particular kind have been included and none have been forgotten.

Predicates should be used to make explicit the meaning of values that are used. A Prolog predicate can be used to indicate that a number represents a turn count, or a particular type of square. Many students wishing to find the centre square would look for the number `5` in a set of empty squares, rather than using

an explicit predicate `centre`, and they would return a number `1` rather than a `corner` square. Figures 3–2 and 3–3 respectively give examples of an explicit and implicit representation.

```
centre(5).
corner(1).  corner(3).  corner(7).  corner(9).

choose_move(Board, computer, move(o, Centre)) :-
   centre(Centre),
   member(Centre, Board). %centre
choose_move(Board, computer, move(o, Corner)) :-
   corner(Corner),
   member(Corner, Board). %corner
```

Figure 3–2: Explicitly choosing centre or corner

```
choose_move(Board, computer, move(o, 5)) :- member(5, Board). %centre
choose_move(Board, computer, move(o, 1)) :- member(1, Board). %corner
```

Figure 3–3: Implicitly choosing centre or corner

Abstraction errors mean that conceptually separate components are not created as structurally independent components with well–defined interfaces to the rest of the program. They make the code less readable and more difficult to change, and they make it more difficult to re–use components of the code.

Structural problems cannot be divorced from the programming language and the rules of good design within that language.

Scattering Tasks Through the Code

A good design will divide the task into components that are as self-contained as possible (Simon, 1973). This minimises the number of interactions that the designer must correctly maintain between the components. In a well structured program, a single conceptual task is performed by a single predicate and the predicates that it calls. It is a mistake to scatter parts of the task among different parts of the program. Scattering tasks leads to unsafe design because it tempts a

developer to modify code for one part of the code for that task without realising its implications for the other parts of the task.

Some programs had decomposed a single task into subtasks which were called from quite different places in the program structure. For example, in some programs it is necessary to translate the representation of a move from a value that the user has typed in to a useful square identifier for making a move. Once this translation task has been completed, there should be no need to translate the identifier any further and the entire translation of the move should be done either in `choose_move/3` or in `move/3`. However, P2 did part of the translation in each.

Repetition of Code

Difficulty in structuring a solution can reveal itself in identical code being repeated in different parts of the program. In general a single task should be implemented only once. Repeating the same code allows errors and inconsistencies to slip in.

Repetition of code may arise from the failure to generalise a predicate. Instead of using a single predicate to perform a task on a wide range of inputs, different predicates are needed for each input. For example, most students used the same code to detect a potential line by the user and to detect a potential line by the computer. In order to do this, the relevant player (or the player's token) must be an input argument to the predicate. P8's code for detecting potential lines were not paramaterised by player, and so P8 had to use separate predicates to detect the user's potential lines and the computer's.

A variant of this problem is writing predicates that differ in their structure and even in the details of their behaviour, but whose function in the program is identical. P9's code for detecting potential lines was actually different for detecting user's lines and computer's lines, but the same predicates could have been used for both. This can be seen as a failure to simulate at the right level of abstraction. At a low level of abstraction, P9's two pieces of code for detecting potential

lines do behave differently for detecting the user's lines and the computer's lines. The code returns winning squares in a different order from blocking squares. At a higher level of abstraction the order of returning these squares is unimportant. There is a difference in low-level behaviour but in terms of the game there is no significant difference in behaviour between the two pieces of code.

Repetition of Execution

Difficulty in structuring a solution can also reveal itself in the repeated execution of a single task. For example, P9 derived a list of empty squares from a nested lines structure once each in several predicates called from `choose_move/3`. This could have been derived just once and passed to the inner predicates. In this case, the repeated execution is related to the decision about which values should be passed as arguments to predicates. It is usually considered good programming style to compute values only once, then pass them as arguments to other predicates.

Hidden Bugs

Some students had written predicates which were buggy when considered in isolation, although in the context of the program the bug did not surface. Typically the buggy predicate was not called with the inputs that would have caused incorrect behaviour. These errors can therefore be seen as a failure to run the code properly under all conditions, either via a mental simulation (Adelson *et al*, 1985; Adelson & Soloway, 1985; Steier & Kant, 1985) or via an actual test predicate. Students may be unable to devise appropriate test cases (Kant, 1985).

This kind of error includes buggy code that is never actually called, so its bugginess is not noticed. It also includes failing to check the type of a value that is obtained from the human player or to check its range. The code will work so long as the user always gives one of the expected responses, but it is not robust if the human player types in something unexpected.

Mis–use of Data Constructors

Students are required to create data structures that represent parts of aspects of the game. A few low–level data constructors (such as lists) are re–used in many contexts. Relational data structures are defined by the designer.

Prolog predicates typically work on specified data constructors. If the student tries to use the same code in two different contexts then the data constructors will have to be the same in both contexts, even if they refer to two different conceptual objects. This is an abstraction problem. Common structures such as lists may be re–used in any contexts, but as a rule, the relational structures that represent different objects should be given different names. Giving them the same name leads to code that is difficult to follow and may be buggy (e.g. P2).

One significant decision in representing the board and other data structures is whether to use a record (with a fixed number of entries which may be of different types) or a recursive data structure (with a variable number of entries, all of the same type). The basic Prolog data structure is a term with a fixed number of arguments which are accessed by pattern matching. This makes an obvious record structure. The arguments can also be complex terms, including similar terms nested indefinitely deeply, and so it is possible to build recursive data structures such as lists and trees from terms.

Prolog also supports two special notations for lists. Both of these notations disguise the underlying implementation, which is a term with a fixed number of arguments indefinitely nested. One notation `[H|T]` supports the concept of a list as a linear data structure of indefinite size, whose elements are accessible by recursion. The other notation `[a,b,c]` also supports the concept of a list as a linear data structure, but it forces the list to be of fixed size and it makes its elements accessible by pattern matching as well as by recursion. The two notations can be combined for a list whose initial entries are in some fixed pattern but whose tail is not specified.

Prolog syntax therefore neatly blurs the distinctions between fixed and variable sized structures, and between access by position and access by recursion. It is not surprising that some students were confused.

Some students tried to manipulate Prolog terms as if they were also linear data structures of indefinite size. This is possible in Prolog (using the built-in predicate `arg/3`), and it is useful on occasion, but it is awkward and considerably less efficient than pattern matching. In the noughts and crosses program it is never appropriate to do this and a list should be used instead. Students who chose to use a term to represent the board were led either to write large amounts of repetitive pattern-matching code to access the different elements and then treat them in the same way, or else to use a verbose and inefficient recursion with the `arg/3` predicate.

The reverse problem was shown by other students, who attempted to use lists where records were required. They created lists with a fixed number of elements which were of quite different types and which were not intended to be processed recursively. A term is a more readable and more efficient implementation for these.

3.6.2 Design in the Domain of AI Game-Playing

Students were required to design programs in the domain of AI game-playing programs in general and noughts and crosses in particular. They needed to make decisions about game-playing strategies and about AI algorithms to implement those strategies. Algorithm choices in general have a great effect on the design and effectiveness of the program.

This aspect of the problem was clearest when the students had to decide how the program should make a move. Students could choose between deep search methods to look several moves ahead, shallow pattern-matching heuristics that only look at the current state of the board and shallow generate-and-test methods in which each possibility for the next move is made and assessed. Shallow generate-

and-test methods alone are sufficient for a player which can play according to the rules, win on the next move and block the opponent, but to achieve better play they must be combined with heuristics.

The noughts and crosses program was chosen so as not to require enormous skill in game-playing programs. The exercise focused on the design of Prolog programs (see below) in general and specialised knowledge was not required. The specification allowed the student to choose the level of skill at which the player operated, beyond a very basic ability to make legal moves, to win the game in a single move if possible and to avoid losing next move if possible. We can interpret very few of the students' errors as due to their unfamiliarity with the domain of game–playing programs.

Noughts and crosses can be played almost equally well by using a few heuristics as by using a search method with a great deal of look–ahead. We found few errors at this level. Indeed the most common error here was for students to attempt to build an unnecessarily complex look-ahead algorithm that was beyond their skill to implement correctly or use properly.

There is no strategy in noughts and crosses that will guarantee a win from the first move. Students could choose whether to determine the first move by a heuristic or by some search method with an evaluation function. P1 made a futile attempt to implement a full–scale minimax look–ahead from the first move to the end of the game looking for a guaranteed winning move. Minimax can show that a draw is always possible in noughts and crosses, it will ensure that at least a draw is achieved, and it will ensure that the player will win if possible. The problem is that all starting moves against a perfect opponent lead to a draw, and so minimax treats all starting moves as equally good. In fact against an imperfect opponent some first moves are more likely than others to lead to winning games (taking corners and the centre) and these can easily be implemented as heuristics.

AI algorithms are also chosen for more detailed parts of the program. Prolog supports a generate–and–test approach very well (O'Keefe, 1990), but this approach can be inefficient and inappropriate for particular problems. P4 wrote

a generate–and–test predicate to pick out three different items from a list (i.e. generate three items from the list, then test that they are different) when a more efficient algorithm would subtract each item in turn from the list so that the same item could not be selected twice.

Decisions about algorithms that are specific to Prolog (i.e. programming techniques) will be considered in Section 3.6.3.

Some programs showed a failure to realise the effects of decisions about the program's behaviour in the problem domain. For instance, P6's player tries to prevent a win by the user before completing a line of its own. This is a poor strategy — it is better to win immediately! Only if the computer cannot win immediately should it try to prevent the user from winning.

3.6.3 Design in Prolog

It is claimed that Prolog is a sufficiently high level programming language to be considered a design language in itself (O'Keefe, 1990; Kowalski, 1979). Students on this course had not been formally taught any other software design language. We therefore treat Prolog itself as a language in which software designs can be built. In this language, students can build the mechanisms of their programs and test their behaviour. Prolog is a highly interactive programming language in which it is easy for students to write and test each component predicate before combining components into a larger whole.

We consider students' Prolog errors in three main classes: errors in the students' understanding of the Prolog programming language itself, errors in using and implementing Prolog programming techniques, and errors in representing data structures.

Students of Prolog need to know which primitives are provided by Prolog. They are learning to use and to understand Prolog's pattern matching and unification mechanism, and its control behaviour including predicate calls, backtracking and

recursion. They are also expected to know which built–in predicates exist and how to use them.

Kant has pointed out that some of the power behind a design lies in the power of the design primitives that are offered to the student (Kant, 1985). Knowledge of the programming techniques that are available in a language raises the level of design primitives available to the programmer from the level of the language itself to a more powerful level.

The choice of a data structure in the programming language to represent an entity in the problem domain is a key design decision whose effect permeates the rest of the design.

Misconceptions about the Prolog Language

We found only a few design errors related to misconceptions about Prolog primitives. All the programs executed successfully and played the game according to the functional specification, and so the students had at least found some way to use Prolog successfully to express that specification.

Some students were not aware of all the Prolog primitives. P3 was not aware of the comparison operator `==` and used a combination of `var/1` and `=` throughout instead.

An interesting effect of a lack of familiarity with Prolog was revealed in one student's program. P7 was apparently a very competent software designer when using imperative or functional programming languages but he apparently did not feel comfortable with logic programming. He did not use any unification in clause heads and he consistently used the explicit unification operator in a way that was analogous to an assignment operator in a procedural language such as Pascal. He had, in effect, identified a subset of Prolog which he could understand as if it was an imperative language. His program was harder to read, less declarative and less efficient than it needed to be as a result, although it was otherwise well structured and it ran correctly. His problem appears to be an example of

interference from another programming language, perhaps a problem of analogy (White, 1988).

The most striking evidence of students' remaining discomfort with Prolog was in their use of Prolog's one construct for explicit control of execution, the cut (written '!'). Prolog's backtracking behaviour and the behaviour of cut itself were new to most of these students as neither occur in the other programming languages with which these students might previously have been familiar. Both are known to be difficult for novices to learn (Bundy *et al*, 1986; Taylor, 1990; Fung *et al*, 1990). Several students used cuts in such a way that their removal would affect the output behaviour of the program. It seems evident that these students did not have a clear understanding of Prolog's control structure. The image is of students running the code, finding that their predicates could backtrack in some unwanted way and hastily adding cuts to prevent that behaviour.

Misplaced cuts accompanied other design problems, such as badly formed case analyses. The students' use of cuts often reveals a profound uncertainty about which parts of the code are expected to do exactly what. Misplaced cuts reveal that the student has been unable to design a predicate with the right behaviour on backtracking, or is unsure of what the predicate's backtracking behaviour really is.

Misconceptions about Prolog Techniques

Students who are unfamiliar with Prolog programming techniques may invent idiosyncratic solutions for tasks even though standard techniques exist in Prolog. These solutions are more difficult to read than standard techniques. They are also likely to contain mistakes, since idiosyncratic solutions must always be designed from scratch.

Students may use the wrong techniques for a particular a task or they may try to implement the technique from scratch instead of retrieving and using a familiar implementation (Rist, 1991), in which case they may fail to implement it correctly.

Three groups of well-known techniques were frequently not implemented properly. These were:

- a variety of list recursions (Gegg-Harrison, 1989; O'Keefe, 1990) including ones which used Prolog's property of reversibility;
- various case analyses, including badly formed case analyses which had been patched up with cuts to prevent inappropraite behaviour on backtracking;
- the recursions or failure-driven loops that are used to validate inputs.

We consider just one example in detail here. Examples of the other errors are described in Appendix A.

P1's attempt to find the maximum value in a list of integers is a good example of a bad implementation of a list recursive technique (Figure 3-4). This code requires an extra input which must be supplied by the caller and must be smaller than any integer in the list. This code is a variant of a technique for list recursion that works well for computing e.g. the sum of a list of numbers, but it cannot be used to compute the maximum or minimum of the list (O'Keefe, 1990). Instead of using the correct technique, P1 has patched the incorrect technique, resulting in an idiosyncratic implementation that is fragile. Figure 3-5 shows a version in which the recursive technique has been corrected and a missing part of the case analysis has been completed.

Choice of Data Structures

The choice of data structures has a profound effect on the design of the rest of the code. Decisions about data structures are typically made early in the design process. Some data representations can be manipulated by simpler logic than others, and so they lead to code that is more likely to be correct (Spohrer *et al*, 1985). Picking the wrong data structure means that the code must perform awkward manipulations when a better choice would simplify the problem.

Figures 3-6, 3-7, 3-8 and 3-9 describe the four most widely used representations for the board. This choice of representation was very significant since the board,

```
find_max([], X, X).
find_max([H|T], MaxSoFar, Max):-
     H > MaxSoFar,
     find_max(T, H, Max).
find_max([_|T], MaxSoFar, Max):-
     find_max(T, MaxSoFar, Max).

Call: find_max([3,5,1,3,2], -9999, Max)  Result: Max = 5
Call: find_max([3,5,1,3,2], 10, Max)     Result: Max = 10
```

Figure 3–4: Poor recursive technique to find maximum number in a list

```
find_max([H|List], Max) :- find_max(List, H, Max).

find_max([],X, X).
find_max([H|T], MaxSoFar, Result):-
     H > MaxSoFar,
     find_max(T, H, Result).
find_max([_|T], MaxSoFar, Result):-
     H =< MaxSoFar,
     find_max(T, MaxSoFar, Result).

Call: find_max([3,5,1,3,2], Max)  Result: Max = 5
Call: find_max([3,5,1,3,2], 10)   Result: fail
```

Figure 3–5: Improved recursive technique to find maximum number in a list

or parts of it, is used in most of the tasks in the noughts and crosses implementation and so the code must manipulate the board many times and for many different purposes. Table 3–1 compares how easy it is to implement two different tasks using four different representations for the board.

Decisions about data structures may also involve a trade-off, since some data structures are easily manipulated for some purposes but not for others. Table 3–1 shows that no representation is perfect for both tasks.

A *list of squares* is a list with nine elements. Each element represents a square in the board, ordered in rows from left to right.
In this example, each square is an atom. The centre square and one corner are taken.

```
[empty, empty, empty,
 empty, x,     empty,
 empty, empty, o]
```

Figure 3–6: Board representation: List of Squares

A *list of lines* is a list with eight elements. Each element represents one of the lines (three rows, three columns and three diagonals). Each element contains three squares. In this example, each line is itself a list. Empty squares are represented by integers that reflect their position. The centre square (number 5) and one corner (number 9) are taken.

```
[[1, 2, 3], [4, x, 6], [7, 8, o],
 [1, 4, 7], [2, x, 8], [3, 6, o],
 [1, 5, o], [3, x, 7]]
```

Figure 3–7: Board representation: List of Lines

A *nested lines* structure is a term with three subterms. One subterm represents rows, the second columns, the third diagonals. Each subterm contains the appropriate lines, which in turn contain squares.
In this example, empty squares are uninstantiated Prolog variables. The centre square and one corner are taken.

```
board(row(line(One,   Two,   Three),
          line(Four,  x,     Six),
          line(Seven, Eight, o)),
      column(line(One,   Four, Seven),
             line(Two,   x,    Eight),
             line(Three, Six,  o)),
      diagonal(line(One,   x, o),
               line(Three, x, Seven)))
```

Figure 3–8: Board representation: Nested Lines

A *list of rows* is a list with three elements. Each element represents one of the rows. Each element contains three squares.
In this example, each row is itself a list. Each square is an atom. The centre square and one corner are taken.

```
[[empty, empty, empty],
 [empty, x,     empty],
 [empty, empty, o]]
```

Figure 3–9: Board representation: List of Rows

We compare how easy it is to implement two different tasks using four different board representations. The utility of the representation for the task reflects the conciseness, readability and efficiency of the students' best implementation that uses that representation, or, if we could envisage a better implementation than any student actually used, of the best implementation that we could envisage using that representation.

Board Representation	Task	Utility
List of Squares	Recognise a good move	Hard
	Update the board	Easy
List of Lines	Recognise a good move	Easy
	Update the board	Hard
Nested Lines	Recognise a good move	OK
	Update the board	Hard
List of Rows	Recognise a good move	Hard
	Update the board	OK

Note: When the *nested lines* representation is used, updating the board is in fact easy if the student uses Prolog variables to represent empty squares (Figure 3–8). However, using variables to represent empty squares causes design problems in other parts of the program. If any other representation is used for empty squares then updating this board structure is extremely difficult.

Table 3–1: Compare board representations for different tasks

3.7 Examples of Code Comprehension and Design Improvement

In this section, we give three illustrative examples of design improvement to illustrate the problems and issues involved. We find that a short piece of code may contain many design errors and that design errors cascade. Both the structure of the code and its behaviour must be understood. Code must be understood in all three design spaces. To understand and critique the code successfully even in terms of the use of Prolog, the intentions of the programmer in terms of the problem domain and the intended algorithm must also be understood.

3.7.1 Design Improvement

A single piece of code can contain many different design flaws. In Figure 3-10 we present an example of a single piece of code that contains many flaws. P8's two clauses, one for dealing with the computer's opening moves and one for dealing with subsequent moves, show many different structural flaws. Most of the flaws presented here are in the general structure of the software design and in the use of Prolog. We do not consider the game-playing skills. The flaws are:

- Explicit tests are used instead of pattern matching and unification is treated as if it were assignment;
- Complex nested "or" branches are used instead of dividing the code into separate predicates;
- Identical code to produce a message to the user is repeated in both clauses;
- The significance of the numbers that represent move positions is not obvious. The number 5 represents the centre square, the number 1 represents a randomly chosen corner;

- The cuts are not placed in a way that makes their purpose clear.

The objective of this code is to choose the computers' move. The computer always plays o, the user x. The first clause deals with the computer's first move, which is either the first or second move of the game. The second clause deals with all the computer's subsequent moves.
The comments are not taken from P8's original code.

```
choose_move(State, o, Move-o) :-
    move_number(State, Number), % Which move is it?
    ((Number == 1, Move = 1);   % If first move, take corner
     (Number == 2,              % Else if second move
      member(5-x, State),       %      And opponent has centre
      Move = 1);                %      Take corner
     (Number == 2, Move = 5)),  % Else if second move
                                %      Take centre
    write('I move to position '), % Tell the user
    write(Move),nl,nl,
    !.
choose_move(State, o, Move-o) :-
    convert_to_board(State, Board), % Change board representation
    (win_move(Board, Move);         % Win if possible - or
     block_move(Board, Move);       % Block lose if necessary - or
     other_move(Board, Move);       % Make a good move - or
     member(Move-n, State)),        % Take any empty square
    write('I move to position '),   % Tell the user
    write(Move),nl,nl,
    !.
```

Figure 3–10: Student P8's Code to Choose a Move

Correcting all these errors is not a straightforward task. It would involve major changes to the structure of the clauses. A possible re-write, created by hand, is shown in Figure 3–11. (We do not claim that the improvements we have suggested are necessarily the best possible ones.) The corrections that have been applied are:

- To replace explicit tests using == by pattern matching and treat unification properly;

- To replace the "or" branches by separate predicates that identify the different conditions that they represent;
- To reorganise the predicate so that the message to the user occurs only once;
- To use named Prolog predicates to make explicit the significance of the numbers that represent the centre and corners of the board;
- To embed the call to the general-purpose predicate `member/2` within a special-purpose predicate `occupied_by/3` that sets up arguments to `member/2`;
- To put cuts only where they are needed.

This re-write requires many operations. Some of the improvements might be envisaged as straightforward re-write rules. For example, it would be a mechnical task to separate the "or" branches into sub-predicates, or to join together the common code into a single clause that calls a sub-predicate. Other improvements are more complex and require an understanding of the student's intentions in writing the code (Johnson & Soloway, 1984). It is impossible to assign a meaningful name to a predicate without knowing what it is intended to do. To make explicit the purpose of the integer `5` by embedding it in the predicate `centre/1` requires knowledge about the student's intentions, i.e. that the student intends 5 to represent the centre square. It requires a decision to make that knowledge explicit, a method for retrieving that knowledge when it is implicit, and a method for making it explicit.

```
choose_move(State, o, Move-o) :-
        move_number(State, Number),     % Which move is it?
        make_move(Number, State, Move), % Decide what to do
        write('I move to position '),   % Tell the user
        write(Move),nl,nl,
        !.

make_move(1, _State, Corner) :-   % First move
        make_first_move(Corner).
make_move(2, State, Square) :-    % Second move
        make_second_move(State, Square).
make_move(Turn, State, Move) :-   % Later moves
        Turn > 2,
        convert_to_board(State, Board),
        make_later_move(Board, Move).

make_first_move(Corner) :-
        corner(Corner), % First move - take a corner
        !.

make_second_move(State, Corner) :-          % Second move
        centre(Centre),                     % If centre is occupied
        occupied_by(Centre, x, State),
        corner(Corner),                     % then take a corner
        !.
make_second_move(_State, Centre) :-         % else take the centre
       centre(Centre).

make_later_move(Board, Move) :- win_move(Board, Move).
make_later_move(Board, Move) :- block_move(Board, Move).
make_later_move(Board, Move) :- other_move(Board, Move).
make_later_move(Board, Move) :-
       empty_sq(EmptyToken),
       occupied_by(Move, EmptyToken, Board).

occupied_by(Place, Token, State) :-
       member(Place-Token, State).

empty_sq(n).  % symbol for the empty square

centre(5).
corner(1).  corner(3).  corner(7).  corner(9).
```

Figure 3–11: Improved Version of P8's Code to Choose a Move

3.7.2 Interleaved Code Comprehension and Improvement

In our study, we did not observe students while they wrote their programs and we have only the completed code to work from. We cannot therefore determine the exact causal sequence of design errors, and many different sequences can be imagined. However, we can certainly see that combinations of design errors mean that as design errors cascade, comprehension is interleaved with correction and doing one correction leads to a realisation that another part of the code is flawed. In Figure 3-12 we present part of P9's code to choose the computer's move. The purpose of much of the code is obscure. Understanding and correcting this code is considerably more difficult than the previous example. We present a summary of the process by which we understood and corrected this code.

```
must_win(State,Player,Move):-
       find_empty_sq(9,State,List),
       permute([5,3,1,7,9,2,8,4,6],List,List1),!,
       member(Move,List1),
       empty_sq(Move,State),
       move(Move/Player,State,State2),
       win_state(State2,Player).

prevent_win(State,_,Move):-
       find_empty_sq(9,State,List),
       permute([5,4,6,2,8,3,1,7,9],List,List1),!,
       member(Move,List1),
       move(Move/user,State,State1),
       win_state(State1,user).
```

Figure 3-12: Part of Student P9's Code to Choose a Move

First, we needed to understand the behaviour of `must_win/3` and `prevent_win/3`. In order to do this, we looked at the predicates that they call (which are not reproduced here) and the data structures that they create and pass between them.

We found that `must_win/3` calls `find_empty_sq/2` to return a list of numbers of all the empty squares (`List`). The first argument of `permute/3` is a list of all squares in the board sorted into centre, corners and sides, which we hypothesised to be a list of moves sorted according to how good they are. `must_win/3` calls `permute/3` with this list of moves and the list of empty squares. `permute/3` sorts the list of empty squares into the same order as the list of best moves. `must_win/3` picks a member of this list, calls `empty_sq/2` to check that square really is empty, then calls `move/3` to make a tentative move, and calls `win_state/2` to check that this move really is a win. If so, `must_win/3` returns this winning move. Given the calling modes of `member/2` member is being used to return different empty squares in order. This is a generate and test method through empty squares.

At this point we obserbed that the extra test for an empty square is unnecessary, since the original list only contains empty squares.

We then found that `prevent_win/3` does much the same, but it tentatively makes the user's move instead of the computer's. If a user's move would be a win, the computer moves there instead. It has a different list of best places to look.

As a result of this analysis, we can remove the redundant test for an empty square and we notice that the two predicates are now so similar in form that it is worth considering whether they can be combined into a single predicate.

By looking at the way in which the two predicates are called, it appears that the purposes of the two predicates are to complete a line and to prevent the opponent from completing a line respectively. At a high level, we know that we can prevent the opponent from winning in the next move by exactly the same method that we use to detect our own win in the next move. We only need to swap players. We are prevented from doing this because

- `prevent_win/3` refers to the `user` explicitly;
- empty squares are passed into `permute/3` in a different order in the two predicates.

We can fix the first problem by writing and calling a predicate that swaps players. We could have fixed the second problem at a low level by making the list into another parameter to pass into `must_win`. More usefully, however, we considered how the code is being used at a more abstract level and realised that the order of squares is not significant in an immediate win or a block. As a result, the entire call to `permute/3` in either predicate is unnecessary and can be removed.

We also notice that the list of empty squares is being created twice when it only needs to be created once. Finally, the improved code looks like Figure 3-13.

```
win_or_prevent(State, Player, Move) :-
        find_empty_sq(9, State, List),
        member(List, Move),
        win_or_prevent_test(State, Player, Move).

win_or_prevent_test(State, Player, Move) :-
        win(State, Player, Move),
        !.
win_or_prevent_test(State, Player, Move) :-
        opponent(Player, Opponent),
        win(State, Opponent, Move).

win(State, Player, Move) :-
   move(Move/Player, State, State2),
   win_state(State2, Player).

opponent(user, computer).
opponent(computer, user).
```

Figure 3–13: Improved Version of P9's Code to Choose a Move

Here also we make no claims that our sequence of improving code is the only sequence or that we have made the best possible improvements. The order of improvements will vary, there are other improvements that might be made, and criteria for what is an improvement to the design will vary. The points that we wish to illustrate are the similarities between design improvement and design itself:

- design errors must be corrected at different levels of abstraction;
- code cannot be considered in isolation but must also be considered in the context in which it is being called, as part of the design structure;
- that simulation of the code is necessary at different levels of abstraction to detect flaws in the code and interactions between the parts of the code being improved.

The processes of understanding the code and correcting the design errors are interleaved with one another, and correcting designs is not a monotonic process. Tentative improvements to the code are explored and rejected as a result of simulating their effects. Having "corrected" several aspects of the code we may find that we had made some mistaken assumptions about the purpose of the code, or the context in which it is being used.

3.7.3 Interaction of Solutions to Design Problems

In the following example, we show that design improvements can cancel each other out. We consider a part of P9's code to choose a move (a different part from the previous section). In Figure 3-14, P9 has combined two errors. The task of printing out the same message is repeated in the code. The task of finding all the empty squares is executed repeatedly although it would be more readable and more efficent to find all the empty squares just once and then pass them as an argument to the predicates that need them.

If we look only at the code for doing the printing, we might re-design `choose_move/3` so that in each clause it calls a single predicate `write_message/0` to do it (Figure 3-15). If we also try to ensure that `find_empty_squares/3` is called just once, then we need to restructure `choose_move/3` so it becomes a single clause. In that case the printing code will only be called once in that one clause and it becomes unnecessary to define it as a separate predicate (Figure 3-16). So the first improvement to the program, the creation of a separate

```
choose_move(State,computer,Move/computer):-
        must_win(State,computer,Move),
        write('Computer has moved!'),nl,nl,!.
choose_move(State,computer,Move/computer):-
        prevent_win(State,computer,Move),
        write('Computer has moved!'),nl,nl,!.
choose_move(State,computer,Move/computer):-
        any_move(State,computer,Move),
        write('Computer has moved!'),nl,nl,!.

must_win(State,Player,Move):-
        find_empty_squares(9,State,Empties) ...
prevent_win(State,_,Move):-
        find_empty_squares(9,State,Empties) ...
any_move(State,_,Move):-
        find_empty_squares(9,State,Empties) ...
```

Figure 3–14: Student P9's Code to Choose a Move

printing predicate, is made unnecessary by the subsequent improvement. Design improvement is not linear.

3.8 Implications for Detecting and Critiquing Design Errors

Automating an analysis of design errors is fraught with difficulty. It would be very difficult to tell just where in the software design process a design flaw in the code appears. Most of the programs contain many design errors, and errors interact and cascade. A poor design decision often makes the rest of the design process more difficult. The student may move further and further away from the familiar or well defined parts of the problem space and deeper into uncharted territory where the chances of making further bad decisions are greater.

The process of deciding that a bodge exists really has two phases. Firstly, there must be some overall impression that there is something wrong with a piece

of code. Secondly, there must be a detailed confirmation that there really is a problem and proof that the code could indeed be written in some better way. The second phase is occasionally trivial, but often it is not. We are not justified in saying that a piece of code is badly designed unless we can answer this question positively. Having decided that there is indeed a problem, we could then try to diagnose the problem and suggest repairs for the code.

Many design errors are dependent on context. Some errors can be detected locally and on a syntactic level, but many require a deep understanding of the overall structure of the code and the student's intentions in writing parts of it (Johnson & Soloway, 1984). This may require reasoning at several levels of abstraction. For example, it is necessary to know what the code is doing at a high level of abstraction in order to realise that two predicates which look different and even behave slightly differently in fact serve the same purpose and could be combined.

In studying our protocols of design decisions, we found that recognising and correcting design decisions required the following

- understanding the code and correcting design errors are interleaved with one another;
- understanding the code and correcting design errors both require knowledge about decisions in the design, domain and Prolog problem spaces;
- parts of the code cannot be considered in isolation but must also be considered in the context in which they are being called;
- understanding the program and correcting design errors must both be done at various levels of abstraction. The behaviour of the code and its purpose can be considered at many different levels;
- simulation of the code is necessary to understand the students' code, to detect flaws in that code and to check the correctness of proposed improve-

ments to that code. These simulations must proceed at various levels of abstraction;

- correcting designs is not a monotonic process. Tentative improvements to the code are explored and rejected as a result of simulating their effects. Having "corrected" several aspects of the code we may find that we had made some mistaken assumptions about the purpose of the code, or the context in which it is being used.

Our study has revealed the wide range of design decisions made in intermediate students' programs and the complexity of the interactions between those decisions. It reveals a wide range of implementations with considerable novelty. It reveals many design errors, errors which occur at different levels in the design process and which interact closely with one other correct and incorrect design decisions.

A high-level decision about which structures and concepts are to be made explicit will influence decisions about which data structures and programming techniques are needed to realise those structures and concepts. The converse may also be true — a programmer may choose to represent explicitly those concepts and structures which can easily be created and manipulated in that programming language.

Decisions can most usefully be critiqued in the context of the student's intentions in making that decision (Johnson & Soloway, 1984) and of the previous design decisions that have been made. In order to critique a large piece of code we must be able to work out:

- which design decisions have been made at each level;
- the intended and actual structure of the code;
- the intended and actual behaviour of the code.

The code of a program represents directly only the lowest level of decision-making, and the program represents structural information only. Programming

languages such as Pascal do require some behavioural specifications such as data type declarations, but programs written in other languages such as LISP and Prolog do not require that annotations about the actual or intended behaviour of the code should form part of the program.

The remainder of the thesis describes a system that models some aspects of both structural and behavioural design and the interactions between structural and behavioural decisions. The system also models decisions at two levels of detail, the task that is being implemented and the programming language techniques that are used to implement the tasks.

Code structure In terms of the structure of the program, at the higher design level we deal with the tasks that the student is to perform and at the more detailed design level we deal with the student's choice of language–specific programming techniques. Tasks combine both domain–specific decisions and decisions about algorithms.

The proper use of standard programming techniques increases the effective power of the language in which the student is creating the design (Kant, 1985). Programs written using standard techniques are more readable and more likely to be correct than those using idiosyncratic implementations.

The following recognition tasks are needed:

- the identification of parts of the code with the tasks that the code is intended to perform;
- the recognition of the language–specific programming techniques used to implement those tasks.

Code behaviour In terms of the behaviour of the program, at the higher design level we deal with the objects in the problem domain that the student is to represent using data structures, and at the more detailed level we deal with the student's choice of data structures to represent those objects.

The following recognition tasks are needed:

- the identification of variables in the program with objects in the problem domain;
- the recognition of the data structures used to implement those objects.

Decisions about data representations are typically made early in the design process and they affect the choice of programming techniques that implement the code that is to manipulate them. We can therefore use these decisions to investigate and critique the interactions between design decisions.

3.9 Conclusions

Intermediate students have a wider range of design decisions to make than beginners. They are given a less detailed problem specification and they must make many decisions about how to complete the specification. The behaviour of the program is not fully specified, and the students are free to make many choices about data representations.

Our study shows that intermediate students make a wide range of design errors. Intermediate students make relatively few errors that can be attributed to errors of understanding the primitives and behaviour of the programming language. They still make errors that can be attributed to a misunderstanding or ignorance of "how things are done" in that programming language. They also make errors in the design of general algorithms, and even in a simple domain they may make errors when solving problems in the problem domain itself.

We conclude that an automated system to detect and critique design decisions would be useful in teaching software design. This system must take into account design decisions made in terms of the problem domain, in terms of general algorithms and programming and in terms of programming in a specific language

such as Prolog. It must take into account decisions about both the structure and the behaviour of the program, and the interactions between decisions about different aspects. We concentrate on design errors which are bodges, i.e. they create code that is badly designed but produces more or less the right output, but our approach could also be applied to bugs which prevent the code from producing the right output or any output at all.

In the following chapters, we describe our approach to the recognition and critiquing of data structures and programming techniques.

```
choose_move(State,computer,Move/computer):-
        must_win(State,computer,Move),
        !,
        write_message.
choose_move(State,computer,Move/computer):-
        prevent_win(State,computer,Move),
        !,
        write_message.
choose_move(State,computer,Move/computer):-
        any_move(State,computer,Move),
        !,
        write_message.

write_message :- write('Computer has moved!'),nl,nl.

must_win(State,Player,Move):-
        find_empty_squares(9,State,Empties) ...
prevent_win(State,_,Move):-
        find_empty_squares(9,State,Empties) ...
any_move(State,_,Move):-
        find_empty_squares(9,State,Empties) ...
```

Figure 3–15: Possible Improvement to Student P9's Code to Choose a Move

```
choose_move(State,computer,Move/computer):-
        find_empty_squares(9, State, Empties),
        find_move(State, Player, Move, Empties),
        write('Computer has moved!'),nl,nl, !.

find_move(State, Player, Move, Empties) :-
        ... % as rest of must_win
find_move(State, Player, Move, Empties) :-
        ... % as rest of prevent_win
find_move(State, Player, Move, Empties) :-
        ... % as rest of any_move
```

Figure 3–16: Alternative Improvement to Student P9's Code to Choose a Move

Chapter 4

An Approach to Representing Design Decisions

4.1 Introduction

In the previous chapter, we identified requirements for software analysis to support a tutoring system for intermediate programming students. Our study of students' programs demonstrates that the analysis must identify and relate design decisions at different levels and it must identify and relate both structural and behavioural decisions.

In this chapter, we describe our approach to recognising design decisions in students' Prolog programs. We have built an architecture in which design decisions about data structures and algorithms can be represented and recognised. The following sections describe and motivate the declarative knowledge structures that are used in that architecture. Chapter 5 describes how these knowledge structures are manipulated.

For simplicity, the structure of the code and the behaviour of the code are considered separately. The representation of code structure is described in Sections 4.2 through 4.8 and the representation of code behaviour (more specifically, of

	Structure	Behaviour
Problem Domain	Tasks	Domain Objects
Programming Language	Techniques	Data Structures

Figure 4–1: Dimensions for Recognition

data structures) is described in Sections 4.9 through 4.14. For both code structure and code behaviour, two levels of design decision are considered, one more dependent on the problem being solved and one that is specific to the programming language. Language-specific design decisions are understood in terms of domain-specific intentions.

4.1.1 Requirements for an Architecture

The study of intermediate students' Prolog programs identified the following requirements for the architecture:

As well as the structural and the behavioural aspects of the code, there is also a need to distinguish between decisions related to the problem domain and decisions related to the programming language used for the implementation. The terminology for this is shown in Table 4-1.

Code Structure

- At the higher design level we deal with the tasks that the student is to perform. Tasks combine both domain-specific decisions and decisions about algorithms.

- At the more detailed design level we deal with the student's choice of language-specific programming techniques.

The following recognition procedures are needed:

- the identification of parts of the code with the tasks that the code is intended to perform;

- the recognition of the language–specific programming techniques used to implement those tasks.

Code Behaviour At the higher design level we deal with the objects in the problem domain that the student is to represent using data structures, and at the more detailed level we deal with the student's choice of data structures to represent those objects.

The following recognition procedures are needed:

- the identification of variables in the program with objects in the problem domain;

- the recognition of the data structures used to implement those objects.

Tutoring systems such as Apropos, Talus and SCENT take an approach in which the tutor is given a known task to perform and must determine which programming techniques were used. Our problem is a step more complex because we must also infer which tasks are being performed by the code.

In order to critique the structure of the program, we must identify parts of the program with particular tasks and work out the function of those tasks within the overall program (Williams, 1984). We need to know how parts of the program behave and how that contributes to the program's function. We analyse both the behaviour and the structure of the code. Only then can we critique the choice of that particular component and the way it has been connected to others.

Analysis by Synthesis Given the large number of permutations of the design, we cannot explicitly list all of the possible implementations of the students' code (Johnson & Soloway, 1984). Instead we need a more abstract way to describe,

recognise and critique programs. We use analysis-by-synthesis to map between the abstract representation of the code and the code itself. Code is synthesised as the matching proceeds. The abstract representation of the code describes the tasks that the code performs and the programming techniques that it uses, and so the analysis can be derived as the synthesised code is successfully matched to the student's program.

A problem in analysis-by-synthesis is how the analysis system knows whether the code that is synthesised is correct or buggy. In Proust, only certain pre-specified plans may be applied to each goal, and each plan is labelled as either a correct or a buggy way to fulfill that particular goal. The plan selection and labelling of plans as correct or buggy must be done by the system builder. This limits the ease with which Proust may be extended or applied to new domains, since each time a new goal is added to the system the system builder must specify exactly which correct and buggy plans may be applied to that goal. It also limits the ease with which Proust can be validated. The system itself cannot check whether the system builder has supplied a valid specification of a plan to fulfill a goal.

We improve this approach by not specifying in advance which plans may be used to fulfill which goals. Instead, plans are selected dynamically to fulfill goals. There is a general problem of how to determine whether a plan correctly fulfills a goal, or indeed whether a program fulfills its specification. This is an open question in progam synthesis. We attempt a partial solution with two elements:

- explicit declarations about the roles of sub-goals within plans such that a plan can only be chosen for a goal if its sub-goals correspond to appropriate roles in the plan;

- behavioural constraints on the code that is created, i.e. that the data types must be consistent.

Partial matching of synthesised code with the student's code controls the further synthesis of code and prevents inappropriate code from being generated, thus saving on search.

We have not supplied a mechanism to explicitly synthesise code that is is buggy. Novel and erroneous implementations are both treated as variants on correct implementations.

Novel and erroneous implementations Students' programs contain novel and erroneous implementations. A purely analysis–by–synthesis approach does not work well if one of the components of the student's composed predicate is novel or otherwise cannot be recognised. It must also be possible to work backwards from the student's predicate to its components, to identify those components which are familiar and to isolate the other components. Novel and erroneous implementations lead to many sources of uncertainty in our analysis and many interacting assumptions between hypotheses. The system aims to build up a consistent interpretation of the student's program that can be critiqued.

Given the large number of novel and erroneous implementations, the many design decisions in this program and the wide range of options for each decision, we aim only for a partial understanding of the student's code. We do not expect to analyse the whole of a program of the size of the game–playing programming exercise.

Generality A general objective for the architecture is that it should not be limited either to the game–playing domain nor to the programming techniques used to implement programs in that domain. The architecture should be suitable to include a wide range of programming techniques, and it should be easy to add new programming techniques to it. It should also be modular so that it can easily be adapted to different problem domains.

4.1.2 Outline of Approach

Code Structure Domain–specific decisions and decisions about algorithms are represented in a hierarchy of task and sub–task specifications. The language–specific programming techniques that implement those tasks are incorporated

into *prototypical predicates*. An analysis–by–synthesis approach is used in which a procedure matches tasks to suitable prototypical prototypes and synthesises suitable Prolog code for that task.

The necessary recognition procedures are the identification of parts of the code with the tasks that the code is intended to perform, and the recognition of the language–specific programming techniques used to implement those tasks. These two procedures are closely entwined and are described together in what follows.

Code Behaviour We consider only the behaviour of the program that relates to data structures.

The necessary recognition procedures are the identification of variables in the program with objects in the problem domain, and the recognition of the data structures used to implement those objects. These two procedures are treated separately. *Polymorphic type inference* is used to represent and infer the data structures that are used to represent domain objects. We have devised a representation for objects in the domain that connects the data type inferred for a variable with the representations for that domain object.

Novel and erroneous code Novel and erroneous implementations are first identified by their context. Successful recognition of surrounding code creates *hints* about the programmers' intentions in the unfamiliar code. A further analysis method, *clausal split*, is used to break down the unfamiliar code into smaller components. If the split is successful then the system confirms the programmer's intentions by matching the familiar components. The unfamiliar or erroneous components are isolated and can be critiqued.

Generality The system is designed to be modular. Information about the design is separated from the process of recognition. Design information is represented by data structures that can be removed from the system, or incremented, without affecting the search strategies that are used for recognition. Information

about problem-specific tasks is separated in the task hierarchy from information about the programming language that is represented in the prototypical predicates, so that either may be replaced independently.

4.2 Understanding Code Structure

We understand student's code in terms of a model of program development in which students select and combine domain-independent programming methods in order to perform the tasks that are necessary for that program. Our representation distinguishes between programming *tasks*, which determine *what* is to be done, and programming *techniques* which offer ways to implement the task in the chosen programming language.

We have devised representations for tasks and techniques that can be used for analysis-by-synthesis. Our representation must be capable of representing a wide range of tasks and programming techniques. It must be flexible enough to represent programs that conform to recognisable techniques and also to "one-off" programs that do not conform to any well-known programming techniques.

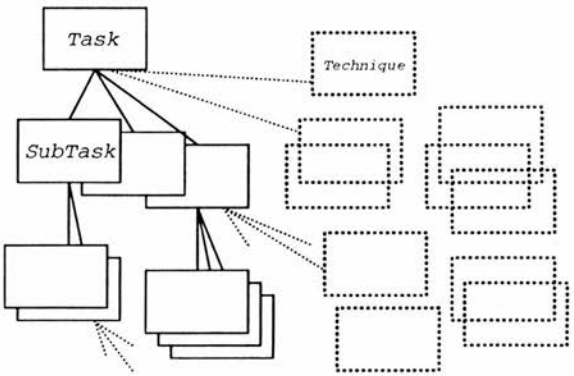

Figure 4–2: Representation of Tasks and Techniques

Tasks and techniques are both ordered into abstraction hierarchies (Figure 4-2). The notion of abstraction is different in each. The task hierarchy refines tasks into sub-tasks, whereas the techniques hierarchy refines general programming techniques into more specific instances of those techniques. Code which is closely related uses similar, combined or refined techniques. The task hierarchy is analogous to the goal-plan hierarchies that are used in Proust (Johnson & Soloway, 1984), whereas the techniques hierarchy represents the derivation of the techniques and their relationship to one another. It is similar to the abstraction and aggregation hierarchies used in the SCENT tutor (Greer *et al*, 1989).

The representation expresses the relations between tasks and the techniques that can be used to perform those tasks. Some choices of tasks and techniques interact, e.g. some sub-tasks are not required for all the prototypes that might be used for the overall task.

The following sections deal with various aspects of representing code structure. Section 4.3 describes the domain-specific representation and Sections 4.4 and 4.5 describes the Prolog-specific representation. Section 4.6 describes how both domain-specific and Prolog-specific representations are combined during synthesis, Section 4.7 describes how the results of the analysis of code structure is represented. Section 4.8 discusses the issues in designing a representation for Prolog programs.

4.3 Representing Tasks

We wish to make an analysis in which the domain-specific part of the design is separated from the language-specific part. We make an initial attempt to encapsulate the domain-specific part using a structure of tasks and sub-tasks.

Each task is represented in the system by a *task definition*. A task definition is an unordered set of *sub-task descriptions* plus a *task header*. A sub-task description

consists of two parts — a *role description* and a *body*. The role description dictates how the body is matched into a prototype. It is a schematic expression which shows how the symbols in the body act as parameters. The body is either a piece of Prolog or a *task specification*. A task specification is a schematic description which could match the header of another task. The task header consists of a name plus the number of domain objects the task manipulates as inputs to or outputs from the task. Some task definitions also contain a domain object specification with further information about domain objects, and this is described in Section 4.13.

The system is supplied with a task definition for each task that it is expected to recognise. An implementation of a task is created by selecting an appropriate prototype and substituting the sub-tasks into the roles for that prototype. The role description for a sub-task is matched to a role slot within a prototype. The body of the sub-task is substituted for the role slot within the prototype.

Variables are local to each sub-task description. The names of role descriptions have universal scope in the system. For a sub-task body to be substituted into a prototype, the role descriptor for the sub-task must have literally the same name as the role slot in a prototypical predicate.

Our representation is an alternative to the goals and plans of Proust (Johnson, 1990). Our tasks correspond to Proust's goals, our sub-tasks to sub-goals, and our prototypes correspond to Proust programming plans. We aim to represent non-programming plans, not as a higher level layer of goals and plans which are then refined into programming plans, but by grouping together all the sub-goals of a goal regardless of the programming plans in which those sub-goals might be used. We present a goal-plan analysis in which goals have sub-goals that are largely independent of the programming plan that is used to fulfill them.

For any task, some sub-tasks may not be required for a particular implementation. Which sub-tasks are included in an implementation is determined by the choice of prototypical predicate to implement the task.

Sub-task Descriptions	
Role description:	Body:
$\mathcal{B}$ (Base)	`Base = 1`
$\mathcal{I}$ (I, Next)	`Next is I + 1`
$\mathcal{T}$ (Element)	*empty_square*(Element)

Domain object specification: BOARD
PLACE

Figure 4–3: Example of a Task Definition: Find an Empty Square

In figure 4–3, we illustrate the task structure for the simple task, that of finding the position of an empty square in a list of squares. This task consists of three sub-tasks, $\mathcal{B}$, $\mathcal{I}$ and $\mathcal{T}$, which initialise a counter, increment that counter and perform a test respectively. The first two of these sub-tasks map onto Prolog code. The third sub-task maps to a task specification to test a whether a square is empty, which has a separate task definition.

4.4 Representing Programming Techniques

Tasks represent *what* the student is trying to achieve in terms of the problem domain. Prototypical predicates represent *how* the tasks can be achieved in terms of the programming language. Each prototype consists of one or more prototypical Prolog predicates, which consists in turn of one or more prototypical Prolog clauses.

Some parts of the prototype represent Prolog code that is used for all of the tasks that are implemented using that prototype. Other parts of the prototype are *slots* which refer to the different sub-tasks for each task. The contents of a slot are specific to the task that is to be implemented.

The parts of a prototype that are identical for all the tasks that are implemented are: the head of the clause; recursive calls; calls to other predicates within the same prototype; and calls to system predicates (such as `,` `;` and `!`),

Figure 4–4 shows an example of a prototype to find the position of an element in a list. This prototype is one of several that might be used to implement the task of finding an empty square. The role slots in this prototype can match to the role descriptions in the task definition for finding an empty square as given in Figure 4–3.

Tested Element with Naive Counter

```
p([H|T], Pos)  :-  T (H),  B (Pos)
p([H|T], Pos)  :-  p(T, P1),  I (P1, Pos)
```

Role slots: $\mathcal{T}$ Sub–Task $\mathcal{T}$ tests an element
$\mathcal{I}$ Sub–Task $\mathcal{I}$ increments a value
$\mathcal{B}$ Sub–Task $\mathcal{B}$ returns the initial value of the counter

Figure 4–4: Example Prototype to Find the Position of a List Element

---

The functor in the head of a prototypical clause is generic to all tasks that can be implemented using that prototype. The functors in the body of the clause are either generic to all tasks or else they may be replaced by code that is specific to a task.

The objective of prototypical predicate is to encapsulate programming techniques. In Pascal, a programming language whose syntax makes heavy use of keywords, programming techniques can often be identified by the use of keywords such as the use of a `while` loop or an `if` statement (Johnson & Soloway, 1984). Languages like LISP and Prolog rely far less on keywords, so even quite fundamental programming techniques in these languages cannot be identified using keywords as cues.

There is at present no comprehensive description of programming techniques that might be used in Lisp or Prolog programs. The representations of programming techniques that have been devised for LISP and Prolog deal with only a subset of the programming techniques that have been identified.

The analyses of Prolog programming techniques that exist in the literature each describe programming techniques at very different levels of granularity. The pro-

gramming operations supported by Bundy's recursion editor (Bundy *et al*, 1991) are perhaps the finest grain. The techniques supported by Bowles techniques editor (Bowles & Brna, 1993) are slightly larger scale, but are still at a finer grain than those described by Brna *et al* (Brna *et al*, 1991), Gegg-Harrison (Gegg-Harrison, 1989; Gegg-Harrison, 1991) and O'Keefe (O'Keefe, 1990). In our architecture, each prototypical predicate represents a way to implement a single task. We do not specify the degree of refinement at each step between more abstract and more concrete prototypes. This makes it possible to experiment with different schemes.

Each prototype is represented in the system by a *prototype definition.* A prototype definition consists of one or more *prototypical predicates* plus a *prototype name.* A prototype definition may also contain a *join description* (described in Section 4.5). A prototype definition also contains a note of the name and number of arguments for that top predicate in the definition. The top predicate is called by no other predicates within that prototype. A prototype definition may also be annotated with the data types of each argument to the top predicate. This type annotation is not supplied by the system builder but is inferred automatically from the predicates using the type language described in Section 4.12.

Each prototypical predicate consists of one or more *prototypical clauses.* Prototypical clauses are ordered, because the order of clauses within a Prolog predicate determines the order in which they are executed. Each prototypical clause consists of a *head* and a *body.* The head consists of a simple piece of Prolog, a term with arguments, such as would appear in the head of a Prolog clause. The heads of all the prototypical clauses within a single prototypical predicate have the same name and the same number of arguments. The body consists of one or more term connected by ',' and ';', Prolog's 'and' and 'or' connectives respectively. Each term may be one of three possibilities: a call to one of Prolog's system predicates, or a call to one of the other prototypical predicates within that prototype, or a *role slot.* A role slot is a schematic expression which dictates how a sub-task description fits into the the prototype.

The head and the body of a prototypical clause may contain variables, and the names of these variables are local to a prototypical clause. Each prototype is self-contained, so that a predicate within one prototype cannot explicitly call a predicate within another prototype. The different prototypical predicates within the same prototype definition are distinguished from one another because they either have different names in the heads of their clauses or else they have different numbers of arguments. The names of prototypical predicates have no scope or significance outside that prototype definition, and so predicates from different prototype definitions may have the same name. It is the prototype name that distinguishes between different prototypes. The names of role slots have universal scope in the system.

Matched Element	Tested Element
`p([H|T], H) :- true`	`p([H|T]) :-` $\mathcal{T}$ `(H)`
`p([H|T], X) :- p(T, X)`	`p([H|T]) :- p(T)`

The role of $\mathcal{T}$ is to test an element

Figure 4–5: Two Example Prototypes for Recursive List Processing

Naive Counter	Tail Recursive Counter
`p(C) :-` $\mathcal{B}$ `(C)`	`p(C) :-` $\mathcal{B}$ `(B), q(B, C)`
`p(C) :- p(C1),` $\mathcal{I}$ `(C1, C)`	`q(C, C) :- true`
	`q(C1, C) :-` $\mathcal{I}$ `(C1, C2), q(C2, C)`

The role of $\mathcal{I}$ is to increment a value
The role of $\mathcal{B}$ is to return the initial value of the counter

Figure 4–6: Two Example Prototypes for Recursive Counting

Figures 4–5 and 4–6 illustrate the prototypical predicates for four very simple programming techniques.

The prototypes in Figure 4–5 represent programming techniques to recurse down a list until some condition is satisfied on an element. The element that satisfies the test is distinguished either by comparing it with an input value in the first example or else by some other test in the second example. The details of the test in the second example are left to the task definition. The prototypes in Figure 4–6 represent two programming techniques which can be used for a wide range

of tasks, which we consider here in the context of maintaining a counter. Both of these techniques generate values, starting with a base value and supplying an incremented value on backtracking. One is a naive technique, the other is a tail recursive accumulating technique. In both of the counter prototypes some details of the counter, its initial value, and its increment or decrement, are left to the task definition.

Our prototypical predicates are hand–crafted rather than derived automatically. This gives us a lot of flexibility to decide what constitutes a technique. It allows us to use the same representation for programs that use well–understood techniques and for programs that are one–offs and use no standard programming techniques.

Different prototypes may represent different degrees of refinement of the same program. There is not a clear distinction between what should form part of a domain–specific task definition and what should be considered part of the prototype. This distinction is somewhat subjective. In our prototype to find the position of an element in a list in Figure 4–4, and in the prototypes that represent the component techniques for counters and list recursions in Figures 4–5 and 4–6, we have followed O'Keefe in allowing the base and increment to form part of the task definition. We might equally have created prototypes in which the base and increment have been set to one, since those values are used almost exclusively in by the students in this exercise.

A similar problem of making the right distinction is found in the skeletons, techniques and enhancements of Sterling and Lakhotia's refinement method (Sterling & Lakhotia, 1988). In their terminology, the list processing part forms a *skeleton* and the counter is a *technique* that enhances that skeleton. This distinction is needed in order to synthesise correct code, but the need is not compelling when completed code is being analysed.

The proper distinction between a skeleton and a technique is not always obvious. Sterling and Lakhotia were able to make this distinction easily because they were comparing meta–interpreters which use complex Prolog data structures. In that

context the traversal of the complex data structure is the important part of the computation, more significant than a counter. The traversal of the complex data structure is therefore treated as the skeleton. By contrast, a counter and a list structure are both simple recursive structures. Either might be treated as a skeleton or as a technique to be applied to that skeleton. In the context of a program which traverses such a simple data structure as a list, the list traversal and counter are of equal complexity and importance.

Gegg–Harrison has shown that the same program schema may be viewed in different ways (Gegg-Harrison, 1991). For pedagogical purpose Gegg–Harrison has chosen to group his schemas according to the list processing techniques that the schemas use, and to treat the use of counters as a secondary variation. But we have also identified bugs in students' implementation of counters even when they have implemented the list recursion properly. For these reasons, we treat the traversal of lists and the maintenance of counters as programming techniques of equal importance.

4.5 Representing Composed Programming Techniques

Prototypes are implicitly related to one another by the programming techniques that they use and extent to which the predicates have been refined. Different prototypes may incorporate similar programming techniques, and one prototype may be a refinement of another. We have not defined a fixed set of operators by which to relate one prototype to another. Various schemes for relating prototypes based on the techniques that they use might be proposed, based on the operations e.g. (Bundy, 1988), the schemas of (Gegg-Harrison, 1991) or the simple techniques used in (Bowles & Brna, 1993). A scheme based on program refinement might be based on the notion of enhancements and mutations as proposed in (Lakhotia & Sterling, 1987) and used by (Vargas-Vera *et al*, 1993; Fuchs & Fromherz, 1992).

In this work we consider only one relationship between prototypes. That is the composition of prototypes that represent different simple programming techniques into larger prototypes that combine those techniques. We propose *clausal join* as a suitable operator by which composed prototypes may be synthesised (Lakhotia & Sterling, 1987), and the reversal of clausal join, *clausal split*, as a way to reconstruct from the composed predicate the component prototypes that represent the component programming techniques (Bental, 1992). We first describe the join specification which controls both clausal join and clausal split. By way of background, we describe the clausal join operation. Finally we describe how the information about components and compositions is represented within our analysis system. The clausal split operation is described in detail in Section 5.3.5.

A clausal join or clausal split is controlled by a join specification which states how the composed predicate relates to its components, and in particular, how the arguments to the predicates are merged. As described in (Lakhotia & Sterling, 1987), a join specification consists of a target term for the composed predicate on the left-hand side of the operator ($\Leftarrow$) and an operand term for each of the component predicates on the right-hand side. Operands on the LHS are connected by the join operator ($\bowtie$). The arity of each term is the same as the arity of the corresponding predicate. The join specifications use the same operand names in the join specification as will be used for the components that are to be joined and their clausal join procedure creates a composed predicate whose name is the same as that of the target term on the the left-hand side of the join specification. The names and positions of variables within the target and operand terms indicate a correspondence between variables in the component and composed predicates.

The clausal join procedure combines clauses from each component predicate to give a composed predicate that has the same functionality as would be achieved by performing its component procedures one after the other (Lakhotia & Sterling, 1987). The clausal join procedure is supplied with one

clause from each of the component predicates. The join specification dictates how the heads of the clauses and any recursive calls within the body of the clauses are combined. The head of the composed clause is created by unifying the heads of component clauses with the operands of the same name and transferring the resulting values of variables from the operands to the variables in the target term. This makes the target term into the head of the composed clause. If there are recursive calls to the component predicates within the body of the component clauses then these are also turned into calls to the composed predicate. For each recursive call, a new version of the clause is created and a composed call is created by unifying the recursive calls in the component clauses with the operands of the same name and transferring the resulting values of variables from the operands to the variables in the target term. The target term is then the appropriate composed call. Auxiliary calls are transferred directly from the component clauses into the composed clause.

Sterling and Lakhotia describe several variants of clausal join, including one–one join and procedural join (Sterling & Lakhotia, 1988). The central algorithm for combining clauses is the same for all these variants, but they differ in how clauses are selected for merging. The one–one join merges each clause of one component with just one clause of the other. Sterling and Lakhotia apply the one–one join to components which traverse the same instance of a data structure, so that for instance a clause that deal with leaves of a tree is joined with a clause that deals with leaves, and a clause that deals with branches is joined with a clause that deals with branches. The procedural join merges each clause of one component with every clause of the other. They apply procedural join to predicates that process different instances of the same kind of data structure.

Sterling and Lakhotia restrict the application of clausal join to predicates that have been derived from the same skeleton and are not mutations. In this case the component predicates will have the same control flow. In a one–one clausal join, all the component predicates must have the same number of clauses, and corresponding clauses in each procedure are joined. The composed predicate will have

the same control flow as its components. Two clauses correspond if they were both derived from the same part of the original skeleton. Sterling and Lakhotia claim that information about correspondence might be maintained automatically but in their implementation they manually find corresponding clauses and mark them by storing them in the same order (Sterling & Lakhotia, 1988).

We apply the idea of clausal join to a different domain to that considered by Sterling and Lakhotia. A clausal join can also be be applied to component predicates that recurse on different kinds of data structures and that do not have the same control flow. The resulting predicates may have different control behaviour from the components. Broadening the scope of clausal join in this way makes it more difficult to make any inferences about the properties of the resulting predicate, but such predicates can usefully be joined if the properties of both the components and the composed predicate are known in advance. Then these properties can be supplied to an automated system rather than being inferred by it.

The prototypical predicates in Figure 4–4 combines two simple recursive programming techniques, that is, a list recursion and a counter. There are several variants of this method of finding the position of a particular element in a list, each of which uses slightly different techniques for the list and the counter. Figure 4–8 shows the join specification which combines the list recursive and counting components in Figure 4–7 to give the composed prototype in Figure 4–9. Figure 4–10 shows the join specifications and the resulting composed prototypes for all of the pairs of components in Figures 4–5 and 4–6. Each of the two list recursions in Figure 4–5 can be composed with either of the two counters in Figure 4–6. Different join specifications are needed for each. The join specifications and the four prototypical predicates that result are shown in Figure 4–10. All of the compositions find the position of an element in a list.

If a composed prototype can be split, then the necessary information for clausal split is represented in the *join description* of the composed prototype. The join description for a prototype both specifies how the code that is synthesised from

Matched Element	Naive Counter
`p([H|T], H) :- true`	`p(C) :-` $\mathcal{B}$ `(C)`
`p([H|T], X) :- p(T, X)`	`p(C) :- p(C1),` $\mathcal{I}$ `(C1, C)`

The role of $\mathcal{T}$ is to test an element
The role of $\mathcal{I}$ is to increment a value
The role of $\mathcal{B}$ is to return the initial value of the counter

Figure 4–7: Components to be Joined

$$position(List, Match, Count) \Leftarrow match(List, Match) \bowtie count(Count)$$

Figure 4–8: Example of a Join Specification

Matched Element with Naive Counter

`p([H|T], H, C) :-` $\mathcal{B}$ `(C)`
`p([H|T], X, C) :- p(T, X, C1),` $\mathcal{I}$ `(C1, C)`

The role of $\mathcal{T}$ is to test an element
The role of $\mathcal{I}$ is to increment a value
The role of $\mathcal{B}$ is to return the initial value of the counter

Figure 4–9: Composed Prototype

that prototype can be split and it also identifies the prototypes from which the components might have been synthesised. A composed prototype is only split if each component of the split is also a prototype that is represented in the system. The join description contains one join specification for each prototypical predicate in the prototype definition. The join description also contains the names of the component prototypes.

Matched Element with Naive Counter

$position(List, Match, Count) \Leftarrow match(List, Match) \bowtie count(Count)$
`p([H|T], X, Pos) :- H=X, B (Pos)`
`p([H|T], X, Pos) :- p(T, X, P1), I (P1, Pos)`

Matched Element with Tail Recursive Counter

$position(List, Match, Count) \Leftarrow match(List, Match) \bowtie count(Count)$
$position(List, Match, Acc, Count) \Leftarrow match(List, Match) \bowtie count(Acc, Count)$
`p(List, X, Pos) :- B(Base), q(List, X, Base, Pos)`
`q([H|T], X, Pos, Pos) :- H=X`
`q([H|T], X, P, Pos) :- I (P, Next), q(T, X, Next, Pos)`

Tested Element with Naive Counter

$position(List, Count) \Leftarrow test(List) \bowtie count(Count)$
`p([H|T], Pos) :- T (H), B (Pos)`
`p([H|T], Pos) :- p(T, X, P1), I (P1, Pos)`

Tested Element with Tail Recursive Counter

$position(List, Count) \Leftarrow test(List) \bowtie count(Count)$
$position(List, Acc, Count) \Leftarrow test(List) \bowtie count(Acc, Count)$
`p(List, Pos) :- B(Base), q(List, Base, Pos)`
`q([H|T], Pos, Pos) :- T (H)`
`q([H|T], P, Pos) :- I (P, Next), q(T, Next, Pos)`

Figure 4–10: Composed Prototypical Predicates

4.6 Representing Synthesised Code

Task and prototype definitions are supplied to the analysis system. They are used for automated code synthesis so that the synthesised code can be matched against the students' code. In this section we describe the code that results from the synthesis. The process of code synthesis is described briefly here in order to make the connection between task and prototype definitions clear, and the process is covered in more detail in Section 5.2.1.

Code is synthesised by combining a task definition with a prototype definition. Each role description in the prototypical predicates is paired with a sub-task (the sub-task with a matching role description). The role description in the prototype is replaced by the body of the sub-task, with appropriate variable bindings. A synthesised predicate consists of a combination of Prolog code and some task specifications. If the body of the sub-task description consists of a task specification then the task specification is included in the body of the synthesised Prolog code.

Figure 4-11 shows an example of the representation of a task and a prototype, and of the code that would be synthesised from that task and prototype. Figure 4-12 shows the code that results from the task definition to find an empty square in Figure 4-3 and the four different prototype definitions that might be used to implement it as shown in in Figure 4-10.

The successful combination of a task with a prototype to create code is called a *realisation* for that task. A realisation consists of an identifier, the task header, the prototype header and the synthesised predicates. There may be several prototypes that can be successfully applied to a single task to yield code, and so the system may create several realisations for each task. A realisation may be either a *partial realisation* or a *complete realisation.* The predicates in a complete realisation consist only of Prolog code. They represent a complete implementation of the task. The predicates in a partial realisation contain one

or more task specifications for sub-tasks of the main task. The synthesised code to find an empty square in 4–11 is a partial realisation because it contains a call *empty_square*/2, a task specification that matches another task header. A partial realisation represents part of the implementation of a task. Other realisations in the system are synthesised to represent the different implementations of each sub-task that is referred to in the predicate body.

Domain Task Definition: Find an Empty Square

Role description:	Body:
$\mathcal{B}$ (Base)	`Base = 1`
$\mathcal{I}$ (I, Next)	`Next is I + 1`
$\mathcal{T}$ (Element)	*empty_square* (Element)

Prolog Prototype: Tested Element with Naive Counter

```
p([H|T], Pos)  :-  T (H),  B (Pos)
p([H|T], Pos)  :-  p(T, P1),  I (P1, Pos)
```

Role slots $\mathcal{T}$, $\mathcal{I}$ and $\mathcal{B}$ of the counter

Realisation

Task Name: Find an Empty Square
Prototype Name: Tested Element with Naive Counter

Synthesised Prolog Code: Find the Position of an Empty Square

```
p([H|T], 1) :- empty_square (H).
p([H|T], Pos) :- p(T, P1), Pos is P1 + 1.
```

Figure 4–11: Task, Prototype and Corresponding Realisation

The replacement of a role slot by a sub-task body corresponds to unfolding a called predicate into the body of the calling predicate. The prototypical predicate is the calling predicate, the role slot in the prototype is the call and the sub-task description is the called predicate, with the role description acting as the head of the called predicate and the body acting as the body of the predicate.

There is a very close relationship between the task structure that is input to the system and the calling structure of the Prolog predicates are synthesised from it. Ideally we would like to express all the variants of an implementation in the task definitions and the prototypes, but this is not possible for variants that change the mapping between tasks and predicates. Given the initial realisations, it is

From Prototype: Matched Element with Naive Counter

```
p([H|T], H, 1).
p([H|T], X, Pos) :- p(T, X, P1), Pos is P1 + 1.
```

From Prototype: Matched Element with Tail Recursive Counter

```
p(List, X, Pos)       :- q(List, 1, Base, Pos).

q([H|T], H, Pos, Pos).
q([H|T], X, P, Pos)   :- Next is P + 1, q(T, X, Next, Pos).
```

From Prototype: Tested Element with Naive Counter

```
p([H|T], 1) :- empty_square (H).
p([H|T], Pos) :- p(T, X, P1), Pos is P1 + 1.
```

From Prototype: Tested Element with Tail Recursive Counter

```
p(List, Pos)       :- q(List, 1, Pos).

q([H|T], Pos, Pos) :- empty_square (H).
q([H|T], P, Pos)   :- Next is P + 1, q(T, Next, Pos).
```

Figure 4–12: Four Realisations of the Task: Find an Empty Square

possible mechanically to generate variants from them in which a single realisation performs more than one task. For example, the first example in Figure 4–13 consists of three predicates which realise a task and its sub–task. Two predicates perform the task of finding an empty square and the third predicate performs the sub–task of checking to see whether a square is empty. In the second example in Figure 4–13, the realisation of the sub–task has been incorporated into the realisation of the main task. We consider that for these variations the underlying task structure and programming techniques remain the same. Unfolding is the mechanical process by which the body of a called predicate is incorporated into the body of the caller (Tamaki & Sato, 1984). An unfolding operation could be provided in order to synthesise more variants from the same task structure and provide more flexible matching, although we have not included this in our implementation.

The task of finding an empty square requires a separate sub–task to test a square to see if it is empty. As a result the code calls a separate predicate to test the square.

```
Task find_empty_square
p(List, Pos)        :- q(List, 1, Pos).
q([H|T], Pos, Pos) :-  empty_square (H).
q([H|T], P, Pos)    :- Next is P + 1, q(T, Next, Pos).

Task empty_square
t(empty).
```

Incorporating the realisation of *empty_square* into the realisation of *find_empty_square* results in:

```
Task find_empty_square
p(List, Pos)        :- q(List, 1, Pos).
q([empty|T], Pos, Pos).
q([H|T], P, Pos)    :- Next is P + 1, q(T, Next, Pos).
```

Figure 4–13: Folding Tasks Together

4.7 Representing the Analysis of Code Structure

So far, we have described a representation for task definitions and prototypical predicates that can be used to model design decisions and we have described how that representation can be used to synthesise programs. We now describe how we represent the results of the analysis.

Recognising tasks and recognising programming techniques are mutually dependent. Dealing with completed programs each of which contain many tasks and sub–tasks, we cannot presuppose that we know which tasks each piece of code is intended to perform. Dealing with code written by students, we cannot presuppose that the code correctly implements that task, or that it implements the task using a well–known technique. We need to recognise the technique in order to identify the task, as we can only be certain that code satisfies a particular task if it matches, or is an equivalent variant of, some known technique. We need to recognise the task so as to know which programming techniques might

be intended, and we need to recognise the techniques so as to know which tasks it might be intended to perform.

Recognising tasks presents a problem for programs which are large enough and specified at a sufficiently general level to exhibit variety in their task structure. We do not expect to identify all of code in the program with a particular task. We may identify it as part of an overall task, e.g. as part of the code to choose the next move, but we may be unable to fit it that part into any known structure. That is, we may be uncertain about just what a particular piece of code is for.

Once it is known what task is being implemented, it is possible to comment on how well the implementation has been designed and whether it uses techniques that are appropriate to that task.

We expect only sparse success in matching. Many parts of the student's program may not correspond to the task structure as we have it. A purely top-down approach to the analysis does not succeed when we cannot find matches for high level tasks, even if the implementation of lower level tasks matches. As a result, we do not follow a simple top-down strategy to synthesise and match tasks. Instead we identify tasks independently of one another in the code, and then build up a consistent interpretation from the tasks using the constraints imposed by the task structure. This corresponds loosely to a blackboard method used for the interpretation of sparse and noisy data, which is discussed further in Section 5.1.

Code for each task is synthesised and matched independently of code for any other task or sub-task. Trivial sub-tasks are incorporated into the body of the synthesised code and matched. More complex sub-tasks are treated as independent, and although they impose constraints on the matching these constraints are merely noted and not checked until later. When a set of tasks has been recognised within the code, the constraints between tasks and sub-tasks are checked for consistency.

The result of the analysis is an *interpretation.* The parts of an interpretation that are derived from the structural analysis consist of a set (in the worst case,

an empty set!) of *recognition graphs* plus a set of *hints*. Each recognition graph is made up of one or more *recognition nodes*, which are connected by *link descriptors*. A recognition node is a mapping between one or more of the student's Prolog predicates and a realisation. A recognition node therefore contains the student's predicates, and (from the realisation) it also contains the task header and the prototype name that were used to synthesise the matching predicates, and the predicates themselves. A recognition node may also contain some associated data structure information: it may contain the names of domain objects in the recognised task, and if so it will contain the corresponding data types for those objects as inferred from the student's program (Section 4.13).

A recognition node is either complete or partial: complete if the realisation for that node is complete, partial if the realisation for that node is partial. *Link descriptors* appear in all partial recognition nodes. There is a link descriptor for every task specification in the node, i.e. for every sub–task referred to in the node whose implementation must be found elsewhere. A link descriptor consists of the task specification (i.e. the necessary sub–task) plus the functor of the Prolog call in the student's code which has been matched to that task specification. If there exists a recognition node whose header matches that task specification (i.e. the sub–task has been identified), then the two recognition nodes are linked [1]. Each recognition graph is a maximal set of linked recognition nodes. A recognition graph is complete if none of its nodes contain a task specification for which there is no recognition node. Otherwise it is incomplete.

If an analysis is completely successful, then the interpretation consists of a single recognition graph in which every one of the student's predicates was part of some recognition node. In this case, following the recognition graph from top to bottom could be viewed as corresponding to a top–down program refinement

---

[1] An interpretation may contain inconsistencies between different link descriptors or between a task specification in the link descriptor and a task header in the linked realisation. This is discussed in Section 5.2.3.

process of choosing a task, identifying its sub-tasks, choosing and combining techniques to implement them, and then either putting implementations of very simple sub-tasks directly into the techniques or else repeating the process for each sub-task that is complex enough to require a separate implementation.

If the student's program is incomplete but otherwise contains a correct and familiar task structure and if it uses correct and familiar implementations then the analysis would result in a single incomplete recognition graph. If the student's analysis contains some buggy or novel implementations then we would expect an incomplete analysis that consists of several unconnected recognition graphs, some of which may be incomplete. An incomplete analysis may result from the inclusion in the system of only some of the necessary tasks and prototypes.

If the results of the recognition are sparse, we expect an interpretation to consist of several incomplete recognition graphs. These incomplete graphs may hint at how other parts of the student's program should be interpreted – that particular calls in the student's code should correspond to particular tasks, even though the code itself matches no known implementation of that task.

Hints are derived from the link descriptors. A *hint* is a special case of a link descriptor, i.e. a task specification plus the functor of the Prolog call in the student's code which has been matched to that task specification. In the case of a hint, there is no realisation linked to the task specification. It is hypothesised that the Prolog predicate that is called in the student's code is likely to be a bodged version of the specified task. Hints are used to drive further analysis of the unrecognised code and especially to isolate bodged implementations of Prolog techniques, as described in Section 5.3.1.

4.8 Discussion

Three important issues arise in specifying the task and prototype descriptions to analyse a program. These are:

- Dealing with variations in the task structure;
- Deciding how task-specific each prototype should be;
- Synthesis of correct code.

4.8.1 Variations in the Task Structure

For the purposes of this thesis, we have made a simplifying assumption that each task consists of a single set of sub-tasks which may be combined in different ways, according to which prototype is chosen to realise that task. The breakdown into tasks and sub-tasks is intended to be largely independent of the implementation, although we do not insist that all sub-tasks appear in all implementations of a task. This works well when different prototypes really do represent only minor implementation variations of similar tasks.

This breakdown is less helpful when there are many variants in the task structure. When there are many different ways in the problem domain to break a task into sub-tasks, then the result is that each task definition may have many sub-tasks, and few of the sub-tasks are common to most implementations of the task. We also find that there may be several clusters of very similar prototypes that are used to implement a single task. The prototypes in one cluster have very similar role slots and represent minor implementation variations. The prototypes in different clusters have very different role slots and they are associated with very different ways to break the down the problem at the domain level as well as at the language level. In the current implementation no such distinction is made.

A further development would be to group together some of the sub–task definitions for a single task. This would create an explicit range of domain–specific (i.e. non–programming) plans for the task. The prototypes that have similar sets of roles could also be clustered together. Different non–programming plans could be identified with different clusters of implementation prototypes. This would clarify the relationship between between domain–specific and programming language–specific design decisions.

4.8.2 Specificity of Prototypes

There is a trade-off between the specificity of a prototype and the range of tasks to which it can be applied. Some prototypes can be used to implement a large variety of tasks. Others are defined very closely to the tasks that they implement. A prototype contains at least one role slot that can be filled in by different sub–tasks to perform different tasks. Decisions remain about how specific each prototype should be, that is, how much of each prototype definition should consist of code and how much if it should consist of role slots that are completed by a task definition. This depends to some extent on the range of tasks to which a given prototype is likely to be applied. For instance, our prototypes for counting have role slots that allow different bases and different incrementing steps for different tasks. In fact for all the tasks in the noughts and crosses program the base is usually 1 and the increment is 1, so these could be built in to the prototype definitions.

Ideally, one task definition would be sufficient for all the prototypes that implement that task, and one prototype definition would apply to all the tasks that can use it. In reality, this ideal cannot be achieved, because there is inevitably a relationship between *how* a task is implemented and *which* sub–tasks are required. The sub–tasks descriptions that are presented in the task definition will vary according to which role slots are specified in the prototype definitions.

4.8.3 Correct Combination of Tasks and Prototypes

It is difficult for an automated system to tell whether a prototypical predicate can be used to generate code that will fulfill some task, and will not generate something meaningless. In a program synthesis system this question is critical, since a system that generates meaningless or inconsistent code will be useless. The question is also important in analysis-by-synthesis. If the system generates incorrect code which matches the student's code and if it has no means of telling incorrect from correct code then it will assume that the matched code is correct.

If we restrict the range of techniques that can be applied to tasks too tightly then we will fail to generate some valid implementations, whereas if we apply all techniques to any task then we will generate many implementations that do not actually fulfill the task. Ideally, we would like to generate all and only correct implementations, but automated program generation on this scale is still an open research problem. In order to match tasks to techniques and implementations for those tasks, we need to decide which prototypes can be used to implement a given task and which sub-tasks should correspond to which slots in the prototype.

One possibility is to create by hand the connections from the prototypical predicates to the tasks that they can fulfill. If all the connections between tasks and prototypes have been specified correctly then code that is synthesised from them will be correct. This approach is taken in Proust. In our system, some control of code synthesis is applied explicitly, in that we use a naming convention for roles to match the sub-tasks with the right slots in the technique. One difficulty is in the need to declare all the connections, which imposes a heavy burden on the person who is supplying the tasks and prototypes to the system. When a new prototype is added, it must be connected to all the places in the task structure where it may be used. A further difficulty is in verifying that the connections as specified are indeed correct and will not lead to buggy code.

Ideally, a language is needed to describe the required behaviour of tasks and sub-tasks and the actual behaviour of techniques (and prototypes). The required behaviour of tasks can be matched with the actual behaviour of prototypes to en-

sure the the synthesised code fulfills the required behaviour. Such a behavioural description might include data types, modes, and failure behaviour. A complete language for code synthesis is beyond the scope of this project, but our system uses one aspect of code behaviour to dictate how code is synthesised. Tasks and prototypical predicates are matched partly through checking the consistency of data types. Data type checking in our system ensures that a prototype cannot be applied to a task in a way that leads the resulting predicates to have inconsistent data types. More strictly, a sub-task definition can only be matched successfully to a role slot if the matching and substitution of the sub-task body leads to code that is correctly typed.

Buggy code can be generated by applying the "wrong" prototypes to the tasks. It is possible to make explicit incorrect connections or mal-rules (Moore & Sleeman, 1988) between tasks and prototypes to capture common bugs. It is also possible to generate code from prototypes that are abstractly related to the correct ones (Greer & McCalla, 1989), or to create random bugs by connecting tasks with just any prototype. Our architecture allows us to experiment with creating explicit connections and could be extended to generate code from closely related prototypes. We have not experimented with bug generation.

4.9 Understanding Data Structures

Previous sections have discussed how we represent the structure of the code and the design decisions that relate to the code structure. We now turn to the problem of representing the behaviour of the code, specifically the representation of the design decisions that relate to the choice of data structure.

In order to understand a program, it is necessary to know which concepts in the problem domain are represented by the data that the program handles (Biggerstaff, 1989; Letovsky, 1988). In order to critique the design of the program, we also need to know how these concepts have been represented in the

chosen programming language, and whether the chosen representation is being manipulated in an effective way by the code.

We look at data structures from two points of view, one that is specific to the problem domain and one that is specific to the implementation. Understanding the concepts that have been represented by data structures in the program corresponds in data structures to the recognition of tasks in the code structure. Understanding the data structures that have been used to implement those concepts corresponds in data structures to the recognition of techniques in the code structure.

The implementation-specific view looks at how domain objects may be realised in a Prolog program as a Prolog data structure or part of a data structure. These choices consist of whether to use a Prolog term or a list, the arity of a term and how deeply the arguments of a term or elements of a list are nested.

Design decisions about data structures are complex in terms of both the problem domain and the implementation. Different programs make use of different domain objects and different implementations of those domain objects. Some domain objects were realised in all the students' programs (e.g. the board), but some game objects were realised in some students' programs and not in others (e.g. a count of the moves played so far).

Different domain objects may be combined into a single data structure. Conversely, the same domain object can be realised more than once in the program and in different ways. For example, the students were instructed to use a single variable to represent the state of the game. Most students considered that the domain state was adequately represented by the state of the board alone, one student used a more complex game state which included two different representations of the board and also a move count.

In Prolog programs, data structures are contained in the arguments of predicates. They may also be obtained from the user or from external files, or presented to the user or stored in external files, and they may be stored in Prolog's internal database. The distinction between code and data can be blurred, in that data

structures can be created, stored and then run. We focus on analysing the data structures that appear in the arguments of predicates. We consider that domain objects are realised in the arguments to the predicates that make up the code structure.

The following sections describe and motivate the representation of data structures. Section 4.10 describes the representation of domain-specific decisions about data structures. Section 4.11 motivates our design of a type language and inference mechanism to describe Prolog-specific data structure decisions and 4.12 describes the type language. Section 4.13 describes how the analysis combines these aspects to form an interpretation of the data structures in the program. Section 4.14 discusses some of the issues raised by the use of a type language.

4.10 Representing Domain Objects

The concepts in the game-playing domain are described in terms of a network of related *domain objects*. A *domain object* is a data item in the game, such as the state of the game, the board, the current player and the player's token. Domain objects may be related to one another by a *part-of* relation. For example, the board typically consists of parts which are squares.

The kinds of domain object that are expected in a particular problem domain (in this case, a game of noughts and crosses) are input to the analysis system. Each kind of domain object is represented by a *domain object definition*, which consists of a domain object name plus one or more *domain object descriptions*. A domain object description specifies one way in which domain objects of that kind (e.g. boards, or lines, or squares) may be represented. Each domain object description consists of a name plus a *type constraint* plus zero or more *part constraints*. For example, Figure 4-14 shows schematically the domain object definition for the board in a game of noughts and crosses. Three representations for the board are shown: a list of squares, a list of lines, or a term containing eight lines. The list and term are type constraints while the square and line are part constraints.

The type constraints refer to the language–specific implementation of the domain object, and the permitted range of type constraints depends on the type language that is being used for that application. The type language is described in more detail in Section 4.12. The type constraints used to interpret the programs in this thesis are a *list*, a *term* with some specified non–zero number of arguments[2], an *integer* or an *atom*. If the type constraint is a list, then there must be one part constraint which applies to all elements of that list. If the type constraint is a term, then there must be one part constraint for each argument to the term. A part constraint is the name of another domain object whose representation forms a part of the original domain object. Thus in Figure 4–14, a line domain object may form part of a a board. Figures 4–15 and 4–16 illustrate the domain object definitions we use for lines and squares respectively while Figure 4–17 illustrates the domain object definitions we use for the move and its parts. Figure 4–18 summarises the type and part constraints for these domain objects.

A program may contain many *instances* of each domain object. Different instances of the same domain object may appear within a single data structure. For example the representation of the board as a term of lines implies that there are eight instances of a line in the board. A program may also contain more than one *realisation* of the same domain object. The same noughts and crosses board may be represented in different ways in the program, using different representations for different tasks.

Our analysis does not distinguish between different instances and different realisations of a domain object. As a result, a question arises as to whether all the instances of a domain object in a program should be expected to use the same representation. It is possible that the same data structure in a program could be interpreted as representing one of two (or more) domain objects, and that the same Prolog code could be interpreted as being a realisation of one of two (or

---

[2]We also refer to a term with two arguments as a *pair* and to a term with three arguments as a *triple*.

more) different tasks. If two tasks use the same domain object, and the domain object is apparently represented differently in either case, then this suggests that perhaps one of the interpretations of the task is wrong. In our analysis the inference process comments on any domain objects that appear to be represented differently in different parts of the program, but does not reject an interpretation if it contains such 'contradictions'.

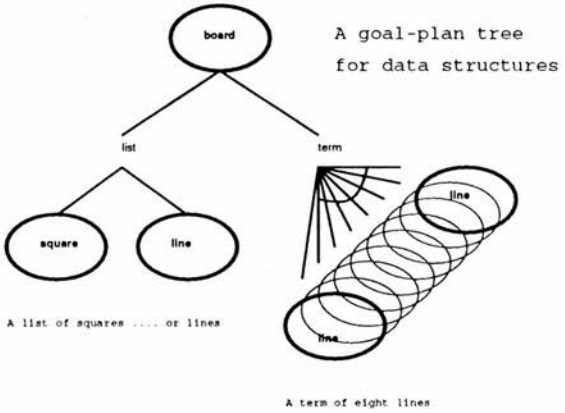

Figure 4–14: Representations of the Board

For instance, in Figure 4–18 one representation for the domain object BOARD is given the name *list_of_lines*, its data type is a list of elements whose type is not specified, and the elements of the list are constrained to be domain objects recognised as LINES.

This representation does not explicitly take context into account. All representations of a domain object are considered equally plausible for all instances of that domain object, no matter which procedure they appear in, or whether they appear as a component of another instance of the same object.

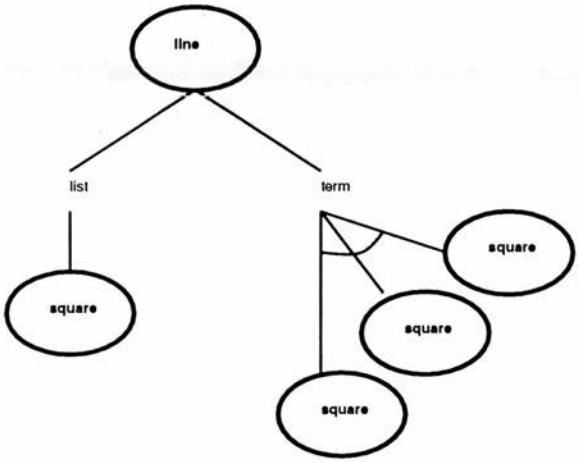

Figure 4–15: Representations of Lines in the Board

4.11 Background to the Type Language

In this section, we consider the problem of describing and inferring the data structures used in a program. Our problems are to determine the set of possible values which an argument to a predicate can take and to describe these sets of values in such a way that they can be related to specific domain objects. This problem is considered in computer science to be the problem of defining and using a suitable *data type* formalism.

In an untyped programming language like Prolog, no data type declarations are required and no type consistency is enforced by the compiler. Yet programs are often written within certain type conventions. In a typed functional language like ML, types are declared for function names. The compiler infers the type of each variable and the type of each value returned by a function from the declarations and the local context and checks that each variable and each function is used in a way that is consistently typed. In typed languages like Pascal and Ada,

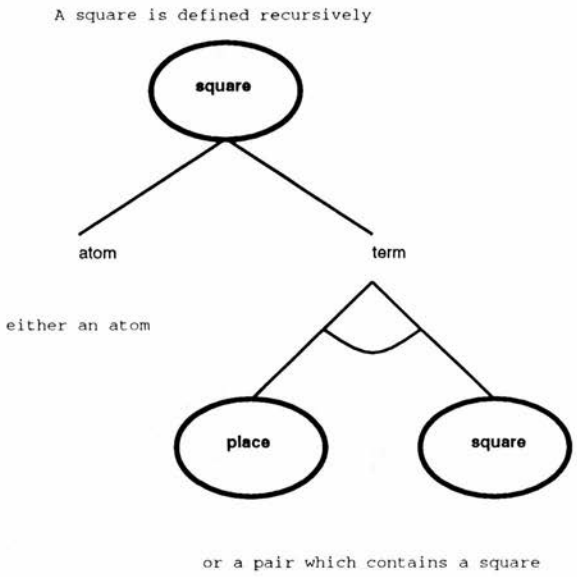

Figure 4–16: Representations of Squares in the Board

the types of variables and function names are defined by redundant declarations, and the compiler checks the consistency of the definition and the ways it is used. When the inference mechanism is provided with so much type information that the inference is trivial, type inference reduces to type checking [3].

Prolog is not an explicitly typed programming language, but Looi has claimed that:

[3]Unfortunately there is no standard terminology in this field. Mycroft and O'Keefe refer to their system as performing "type inference" in (Mycroft & O'Keefe, 1984). It is true that their system infers the types of variables within predicates, unlike Pascal in which the type of each variable is declared. (Hill & Topor, 1992) refer to Mycroft and O'Keefe's system as "weak" type inference, and Mishra's as "strong" type inference.

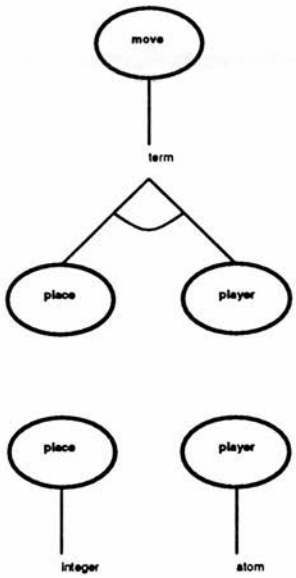

Figure 4–17: Representation of a Move and its Components

> Type inference can be used to infer the types of expressions when little or no information is given explicitly (Looi, 1988b).

Looi successfully uses type inference in his automated Prolog debugger for three purposes:

- to assist in heuristic code matching, by comparing the types of the student's program to the types of the reference programs;
- to set up test data for dynamic analysis;
- to verify that types are used consistently in the student's programs.

Our objectives are rather different from Looi's. Our main objective is to work out how students have represented a particular domain object. Our two other uses for types are to confirm that synthesised code is correct by checking that it is consistently typed, and to speed up search during the code recognition process

Domain Object	Representation	Type Constraint	Part Constraints
BOARD	*list_of_lines*	$list(T)$	T is a LINE
	list_of_squares	$list(T)$	T is a SQUARE
	term_of_lines	$term(T_1 \ldots T_8)$	T_1 is a LINE ... T_8 is a LINE
LINE	*list_of_squares*	$list(T)$	T is a SQUARE
	triple	$term(T_1, T_2, T_3)$	T_1 is a SQUARE T_2 is a SQUARE T_3 is a SQUARE
SQUARE	*atomic_square*	*atom*	
	place_and_content	$term(T_1, T_2)$	T_1 is a PLACE T_2 is a PLAYER
MOVE	*place_and_player*	$term(T_1, T_2)$	T_1 is a PLACE T_2 is a PLAYER
PLACE	*numeric_id*	*integer*	
PLAYER	*atomic_player*	*atom*	

Figure 4–18: Domain Objects and Their Representation

by matching types before trying to do more detailed matching. However, it is our main objective that has influenced the design of the type language and type inference mechanism we have used.

There have been two main approaches to using types in Prolog. Mycroft and O'Keefe's approach to types is prescriptive. Each functor that is used as a data structure must be declared to be of some type, and the types of each predicate's arguments must be declared. Existing typed languages for logic programming are based on this approach, e.g. TEL (Smolka, 1987) and HiLog (Yardeni *et al*, 1992). This approach is mainly applied to type checking. A contrasting approach, and one that is closer to our needs, is taken by Mishra (Mishra, 1984). Mishra's approach is descriptive. He characterises the type of a predicate by a description of the values that will allow a predicate to succeed. This approach is useful for automatic type inference, and in Mishra's proposed system no type declarations are needed whatsoever.

Our approach is type inference, but is not as purely descriptive as Mishra's. A

type inference system that is purely descriptive would not return enough information. In particular, we assume that if an argument is used in one clause with no restrictions on its value, that does not necessarily mean that the predicate is intended to deal with every possible value for that argument. More useful information about intentions can be derived by getting the type information from the other clauses which do restrict the value of that argument and assuming that the same restrictions are intended in all the clauses.

Our type inference is predicate based. The data types for each clause are combined to give the overall type for the predicate. We share with Mycroft and O'Keefe the assumption that a single argument to a predicate has only a single data type. We compute the data type for an argument in the predicate by unifying the types inferred for that argument in each clause.

The language of data types that we use, the type inference mechanisms that we have chosen, and the assumptions that we have made may not correspond to the mechanism, data type languages and assumptions that any student has actually used or intended to use. If the system fails to derive a consistent type for a predicate, that is treated as a failure to derive information with respect to the current set of types and inference mechanism, rather than as a type error on the part of the student.

No complete or ideal set of requirements and features have been identified for a type language that should be used to describe Prolog programs (or logic programs in general). Two requirements that are widely recognised are the need for two forms of polymorphism, *parametric* and *inclusion* polymorphism (Pfenning, 1992). Type systems usually require that when a data type is given (or inferred) for an argument to a predicate, then a properly typed program uses only the values that are described by that particular type, and it does not use values that are described by any other type. This is a rather strict criterion, which is made more flexible and usable by allowing parametric or inclusion polymorphism. Our study of students' Prolog programs has found that to represent

precisely all the data structures that were used, both kinds of polymorphism are needed.

Parametric polymorphism deals with predicates that are called with different types in different calls, yet use the same code to manipulate them. Thus a single predicate `append/3` appends two lists of elements, regardless of the types to work on different subsets of the same set of values, rather than requiring all predicates to work on all values of a particular type. Inclusion polymorphism (also known as sub-typing) allows the values of one type to be a subset of the values of another type. Inclusion polymorphism may be represented by allowing one type to be defined in terms of another type.

Unfortunately, no algorithm currently exists for type inference with both parametric polymorphism and inclusion polymorphism. Dietrich and Hagl conclude that

> ...in general one cannot absolutely determine whether a sub-type relation between two polytypes holds (Dietrich & Hagl, 1988).

Fuh and Mishra (Fuh & Mishra, 1987) have proposed a system which combines type inference, parametric and inclusion polymorphism for functional languages, but this has not yet been applied to logic languages.

Xu and Warren propose an approach which combines type checking and type inference, depending on how much type information is available (Xu & Warren, 1988). They express inclusion polymorphism as well as parametric polymorphism. Unlike the Mycroft-O'Keefe type system, their inference mechanism will cope even if there are no type declarations, although the information it provides will be less useful. However, in their system type declarations are still needed for predicates if parametric polymorphism is to be expressed.

Looi used a system based on Mishra's. Few details of the type inference algorithm are given in (Looi, 1988b) but it would appear from this that Looi did not take advantage of the inclusion polymorphism and used parametric polymorphism instead. Like Looi and Mishra, we use type inference rather than type checking,

and we use parametric polymorphism but not inclusion polymorphism. We have also devised a representation for *anonymous* types which allows us to interpret types with more flexibility.

4.12 A Language to Represent Data Types

We have devised a language that we use to describe data types and implemented a type inference mechanism which is given declarations for data types and for built-in predicates and works out the type of each variable within each of the user's predicates. This approach is based on Mycroft and O'Keefe's polymorphic type system for Prolog (Mycroft & O'Keefe, 1984), but we have used an approach to inference that is more similar to those used for typed functional programming languages (Milner, 1978; Cardelli & Wegner, 1985) and we have extended the language to allow for *anonymous types.*

We have a model of a Prolog program in which predicates are made up of clauses; each clause takes some arguments (i.e. data structures) and manipulates them by calling other predicates with those or other data structures as arguments. We define a data constructor to be the name and number of arguments of a Prolog term that is used as the argument to some predicate. In order to perform type inference, a set of *data type declarations* and a set of *predicate type declarations* are needed.

Type mechanisms have generally been used by program designers who supply the necessary type declarations. Type mechanisms have been used to increase the run-time efficiency of programs, to provide simple and powerful mechanisms for data abstraction and to help programmers write programs whose use of data structures is consistent and are therefore more correct and more robust. Mycroft and O'Keefe's type inference system requires type declarations for all predicates in the program and for all data structures it manipulates. By contrast, we are dealing with completed programs for which the student programmer has written no such declarations. Our task is not to check that a given set of type declarations

is correct for a particular program but to infer what the type declarations should be for that program. Some type declarations are general-purpose and can be supplied in advance, others can be inferred from the program, and still others can only be predicted but with no certainty of success.

We have adapted Mycroft and O'Keefe's type inference system to take an approach more similar to that described in (Cardelli & Wegner, 1985) which infers more types and requires that fewer type declarations be supplied explicitly.

In our type language, a *type* consists of a *type constructor* which has zero or more *type arguments*. Each type argument may be either a *type variable* or another type. We show type constructors in italics and type variables in italic capitals. For example, three possible types are *integer*, $list(integer)$ and $list(T)$. Types that contain type variables are *polymorphic types*, and so $list(T)$ is a *polymorphic type*. Polymorphic types allow us to describe predicates which manipulate data structures that are partially specified. For example, a Prolog predicate can be written to append one list to another to produce a third list. This predicate must operate on lists, but those lists may contain elements of any type. We could therefore specify that the arguments of this predicate are of type $list(T)$. Type variables may be bound to types, so it is possible to combine information about the type of a data structure that is obtained from different parts of the program.

A data type declaration consists of a head and a body, connected by the operator $\leftarrow$. The head of the data type declaration is a type. The body is a set of one or more type specifications connected by disjunction. A type specification consists of a data constructor (i.e. a Prolog functor which may be used as a data structure in the Prolog program to be typed) plus arguments which are themselves either type variables or types. Some examples of data type declarations are shown in Figure 4-19. Data constructors are shown in `typewriter` font.

Each data constructor can appear in at most one data type declaration. This restriction avoids search in the type inference mechanism, which would otherwise quickly become prohibitive.

Data Type Declarations:

list(T)	←	[] ∪ [*T* | *list(T)*]
square	←	`x` ∪ `o` ∪ `empty`
integer	←	*integer* + *integer*

Initial Predicate Type Declaration:

`is`(*integer*, *integer*)

Input Prolog Program:

```
find_empty_square([empty|_], 1).
find_empty_square([_|Rest], Position) :-
        find_empty_square(Rest, P1),
        Position is P1+1.
```

Predicate Type Declarations After Type Inference:

`is`(*integer*, *integer*)
`find_empty_square`(*list*(*square*), *integer*)

Figure 4–19: Illustrative Example of Type Inference

Besides data types declarations, some Prolog predicate types are needed. A predicate type declaration consists of the name of a Prolog predicate plus one type argument for each argument to that predicate.

This data type language language does not allow us to define types with an infinite number of data constructors. For instance we can define the type square with three data constructors `'o'`, `'x'`, and `'empty'` as in Figure 4–19. But the integers as they are normally represented in Prolog have a very large (and theoretically infinite) number of data constructors `'1'`, `'2'`, `'3'`, ... We therefore provide a built–in type *integer* for all integers.

So far, the language we have described is similar to that devised by (Mycroft & O'Keefe, 1984). In the following sub–sections, we describe two important extensions that we have made to their language, type inference and anonymous types. Then we describe the use of this general type language to create a specific set of types for understanding students' programs, and we summarise the inputs and outputs of our type mechanism.

4.12.1 Type Inference

Our system requires only that a predicate type is provided for all the predicates that are built in to Prolog, i.e. system predicates. Predicates defined by the student do not need type declarations. Instead the type declarations for the student's predicates are computed by a type inference mechanism. Mycroft and O'Keefe require that there is a predicate type declaration for every predicate in the Prolog program. Their requirement is suitable for their domain of type checking but is not suitable for our domain of type recognition. Given a set of data type declarations, predicate type declarations and predicates, our type inference mechanism extends the set of predicate type definitions as shown in Figures 4–19 and 4–20. The type inference mechanism is described in more detail in Section 5.7.

4.12.2 Anonymous Types

We provide an extension to the data type language. A set of *anonymous* data type declarations represent the number of arguments in a data constructor but not its name. This deals with the problem of the idiosyncratic names that students use for data constructors. That is we have the types *atom*, $singleton(T)$, $pair(T_1, T_2)$, $triple(T_1, T_2, T_3)$, etc. An anonymous data type has a data type declaration in which the type specifications consist not of a data constructor plus type arguments, but a variable which can stand in for a data constructor, plus type arguments. Figure 4–20 shows examples of anonymous data types. A data constructor is given an anonymous type only if it fails to match a more explicit type. This allows us to recognise data structures at the most specific level of abstraction, as has been done with recognising code structures (Greer *et al*, 1989; Corbett *et al*, 1988).

We have also restricted anonymous data type declarations so that each anonymous data type declaration contains exactly one type specification in its body (as shown in Figure 4–20). This limits the search in the type inference pro-

Data Type Declarations:

$$\begin{array}{lcl} list(T) & \leftarrow & [] \cup [T|list(T)] \\ integer & \leftarrow & integer + integer \end{array}$$

Anonymous Data Type Declarations:

$$\begin{array}{rcl} atom & \leftarrow & P_0 \\ singleton(T_1) & \leftarrow & P_1(T_1) \\ pair(T_1, T_2) & \leftarrow & P_2(T_1, T_2) \\ triple(T_1, T_2, T_3) & \leftarrow & P_3(T_1, T_2, T_3) \\ & \cdots & \end{array}$$

Initial Predicate Type Declaration:

`is`(*integer*, *integer*)

Input Prolog Program:

```
find_empty_square([empty|_], 1).
find_empty_square([_|Rest], Position) :-
        find_empty_square(Rest, P1),
        Position is P1+1.
```

Predicate Type Declarations After Type Inference:

`is`(*integer*, *integer*)
`find_empty_square`(*list*(*atom*), *integer*)

Figure 4–20: Example of Type Inference as Used in Analysis

cess. Unfortunately it also restricts the variables that can be given anonymous types, in that a variable in a predicate can only be given an anonymous type if that variable will only ever contain data constructors that all have the same arity. We have found empirically that even with this restriction much useful type information can be derived from the students' programs.

4.12.3 Choosing Data Type Declarations

We have experimented analysing the student's programs with different sets of data type declarations. Eventually we used only the data types $list(T)$ and *integer* plus the anonymous data types for all our analysis of student programs. We investigated whether a more specific set of declarations would allow the data types to express more domain-specific decisions. People who work together on a professional programming project often use very similar terminology. This

terminology can be obtained using knowledge elicitation techniques and then re-used to understand programs within the project (Iscoe, 1992). By contrast, we found that each student used different terminology in their programs even though they were all working on the same problem. The students worked independently of each other and had no need to develop a common terminology. As a result, we could not reliably supply domain-specific names of data constructors to our system in advance.

For example, it would seem on the face of it that the data type *square* in the illustrative example in Figure 4-19 could be used to identify a specific type *square* that would map closely to the domain object *square*. In the illustrative example in Figure 4-19 we inspected the program to find which values were used to represent the `square`. Unfortunately the students chose such varied names for their data constructors that we could not supply any set of specific types that would work for many different students' programs. More specifically, the problem is that different students used different combinations of constructor names for different data types, so it was not possible to create a single set of types in which each constructor belongs to one type only.

4.12.4 Summary of the Type Language

We have found that a type language can capture the Prolog-specific aspects of data structures that are used generally in many applications. It does not capture the domain-specific aspects of these decisions, which are left to the domain object descriptions as described in Section 4.10. Only when the name of a data constructor is used in a particular way in all Prolog programs, can a specific type declaration which refers to the names and arities of data constructors be usefully supplied. Anonymous types, which describe only the arities of data constructors, are more useful for the other data constructors. Figure 4-20 shows the relevant data type declarations and the results of type inference on a simple program.

The inputs to the type inference mechanism are:

- a set of data type declarations;

- a set of anonymous type declarations;
- a set of predicate type definitions for all predicates that are built in to Prolog;
- some Prolog predicates to be typed.

The end result of type inference process is a predicate type definition for every Prolog predicate that has been input. Our system derives predicate type declarations for the student's program, for the predicates in each prototype definition and for the predicates in each realisation. The derived predicate type declarations are used to infer how the student has implemented domain objects. They are also used during code synthesis and code matching.

4.13 Tasks and the Results of Data Structure Analysis

Some of the tasks in the problem domain are always performed on the same set of domain objects. The top-level predicates in the noughts and crosses program are an example of this. These predicates can be used to derive information about how game objects have been represented. Other predicates (typically found at a lower level) are so general they are performed on all sorts of domain objects, and they cannot be used to derive information about domain objects.

All task definitions contain in their header information about the number of arguments they require. If a task is sufficiently specific so that exactly one domain object corresponds to each argument, then that task is also labelled with the set of game objects that correspond to each of its arguments.

For simplicity, we have assumed that the number of arguments that any task will have is equal to the number of arguments in the top predicate of all realisations of that task. Each argument in the top predicate of the realisation of such a task corresponds to one of the domain objects with which the task is labelled.

The result of the data structure analysis for each interpretation of the program is a set of domain object interpretations. Each domain object interpretation consists of the name of the domain object, the name of a domain object description, and the names of all the tasks and corresponding student's predicates in which the domain object has been found to have that description. Only the tasks that have been matched to predicates in the student's code, and only those tasks that have domain object definitions, appear in a domain object interpretation. Different instances of domain objects in different parts of the program may use different realisations, and so each domain object that has been identified in the program may have several interpretations.

4.14 Discussion

4.14.1 Novel and Erroneous Data Representations

If the inferred data type of a domain object does not match any of the known data representations for that domain object, the system returns no information about the data representation.

Prolog is not a strongly typed language, and so students can write and successfully run programs that are inconsistently typed according to our mechanism. In some cases, we may view this as a limitation on the type inference mechanism that we have used or on the ability of type inference systems in general (Dietrich & Hagl, 1988; Frühwirth, 1988). In other cases, the observation that a data type is used inconsistently can itself be a useful comment on the design (Mycroft & O'Keefe, 1984).

Type inference mechanisms can either treat apparent inconsistencies as errors or else they can treat inconsistencies as failures of the inference mechanism to derive appropriate information about the type. The former is suitable for error detection but the latter approach is more appropriate for inferring the student's intentions as it allows us to derive type information from other sources. Our

system is usually able to obtain some information about the data type that the student intended even when part of the actual implementation uses types inconsistently.

4.14.2 Scope and Limitations of the Type Language

A type inference mechanism has several advantages over just running the program and inspecting the output. Polymorphic data types provides a useful language in which to describe and summarise the data structures created by Prolog programs. The mechanism does not require repeated runs of the program on test data, a major practical issue since the format of input data was not part of the problem specification and students picked a wide range of formats.

Generality in data structures can be expressed along various dimensions. Type variables capture one kind of generality. With type variables it is possible to distinguish between, say, a procedure which creates or manipulates a list of integers and a procedure which works for all lists regardless of the types of their elements. This kind of generality is similar to the goal tree model of hierarchical abstraction that is used to describe functions in the LISP tutor (Corbett *et al*, 1988).

The language has some limitations. It cannot express all the information that we might like to derive about data structures. For example, data structures must be treated either as having finite size or as being infinite. This means that we cannot use this mechanism to distinguish between lists of different lengths which are used for different purposes.

Our type inference mechanism does not in itself solve the problem of mapping between the students' intentions in terms of the problem domain, and the students' implementation in terms of the programming language. A student may use a single data type in the programming language to represent more than one entity in the problem domain. Some structures (such as lists or integers) may be used in several different contexts to represent different entities, and the type inference system cannot distinguish between these different uses.

We experimented with using overloading, in which a single data constructor can be interpreted as belonging to more than one data type. We found that this combines with the lack of explicit type information to give a severe search problem with many redundant and implausible interpretations. Our current system avoids generating multiple interpretations for a single value. Instead, where ambiguity is likely to arise we use the less informative anonymous types.

An alternative approach to solving this problem would be to use a type inference mechanism based on sub–typing (Dietrich & Hagl, 1988; Frühwirth, 1988), in which a single data structure may belong to a more than one data type depending on the context in which it occurs. However, these type inference mechanisms require more starting information about the expected data types than can supplied for a novel Prolog program by anyone other than the designer of the program.

4.15 Summary

In this chapter, we have proposed an architecture for recognising design decisions in students' programs. This architecture is based on the analysis of programs in the previous chapter.

The architecture distinguishes between design decisions that are specific to the problem being solved and with design decisions that are specific to the programming language that is used. The architecture also distinguishes between with design decisions about the program's structure and its behaviour.

Problem–specific design decisions about the structure of the program are represented by a hierarchy of tasks and sub–tasks that must be performed in order to fulfill the problem specification. Prototypical predicates describe the typical ways that such tasks may be implemented in Prolog. From the task structure and the prototypes an automated system can synthesise canonical code that implements different versions of the program, and can be matched against the

student's code to obtain an interpretation of the domain-specific tasks in the student's program and the Prolog programming techniques used to implement them.

The problems of dealing with novel and erroneous code are dealt with by obtaining hints from the surrounding context about the programmer's intentions in the unfamiliar code. Clausal split is used for a more detailed analysis of the unfamiliar code.

We propose a similar division of knowledge structures to describe the data structures that the students' programs create. Domain-specific decisions are represented by the components of domain objects, while Prolog-specific decisions are represented by a data type language. We have investigated the requirements of a type language for recognising data structures in Prolog programs. We have devised such a language, using polymorphism and anonymous types.

The following chapter describes the way in which the architecture manipulates these structures to analyse a student's program.

Chapter 5

A System Architecture to Recognise Design Decisions

5.1 Introduction

This chapter describes the procedural aspects of the architecture whose declarative aspects were described Chapter 4. It first presents the objectives of the system. Then it presents an overview of the procedure by which the system analyses a student's program. The main part of the chapter describes the architectural and implementation details of the analysis procedure.

We have argued in Chapter 4 that in order to critique the structure of the program, we must identify parts of the student's program with particular tasks. The problem of recognising components in a program is comparable to the problem of recognising overlapping objects in a visual scene. Parts of the program which contribute to a single function may be scattered through the program, so we do not know exactly which areas of the picture correspond to which objects. Parts of the program may be written in unconventional ways, so there are parts of the scene that we cannot identify with certainty and even parts that we cannot identify at all. We do not know exactly which objects there are nor how many objects we expect to find.

We therefore have a large piece of structured code from which we attempt to sieve out recognisable components. Having sifted out some components we combine them into larger structures and we identify the function of these larger structures and the function of each component within the structure. This in turn may give us more expectations about some other parts of the code so that we can sieve out still more components and fit them into the structure.

Our approach is similar to that used in the *blackboard model*, which is particularly suitable for this kind of problem (Murray, 1989; Macmillan & Sleeman, 1987; McCalla *et al*, 1986). A blackboard model consists of a range of knowledge-based systems each with a different kind of expertise and all communicating via a shared data area called the blackboard. Each knowledge source may contribute some information to the blackboard. A knowledge source is activated when the appropriate data becomes visible on the blackboard, and it is suspended when it can no longer contribute, until some other knowledge source makes enough data available for it to continue. We have used the idea of independent but co-operating knowledge sources which contribute different kinds of knowledge in which realisations and recognition nodes act as a shared blackboard area.

As in the previous chapter, the description of how code structure is analysed is separated from the description of how code behaviour is analysed. Sections 5.2 through 5.4 describe code structure while Sections 5.6 through 5.8 describe code behaviour.

Section 5.2 describes the analysis-by-synthesis mechanism by which tasks and programming techniques are identified in the students' code. This section describes the synthesis of code from task and prototype definitions, the matching of code to create recognition nodes, the combination of individual recognition nodes into the recognition graphs which represent the results of the analysis, and the creation of hints which drive further analysis. Section 5.3 describes the clausal split mechanism which identifies bodged implementations and localises the bodged code. Section 5.4 describes the use of operators to deal with vari-

ations in the implementations. Section 5.5 describes how the quality of each interpretation of the code is automatically assessed.

Section 5.6 describes how domain objects are identified in the code. Section 5.7 describes the type inference mechanism and its use for identifying the Prolog-specific aspects of data structures, and Section 5.8 describes how the type mechanism deals with the problem of students' novel and incorrect data types.

The remainder of this section presents the objectives and an overview of the recognition process.

5.1.1 Objectives

The specific objectives that our architecture is to realise are as follows:

Identify and combine both structural and behavioural analyses
Combine information from different views of the program to understand it. Use information about what form the code has, how it is called, what data structures the code creates.

Separate problem-specific from language-specific decisions Separate the knowledge that is specific to the problem domain from the knowledge that is specific to the programming language.

Recognise design decisions in a program that includes **novel** design strategies and **buggy** implementations. If a piece of code cannot be understood as a whole, we look for components of the code that confirm our hypotheses about its purpose and we localise as far as possible the components of the code that cannot be understood. These components reflect buggy implementations.

Use partial information Hypotheses are created about the purpose and implementation of the code based on analyses of different sections and aspects of the code. Hypotheses are confirmed or rejected as partial hypothesis are

combined. It is expected that some parts of the code will not be understood fully.

Create a general architecture The architecture should be useful for a wide range of programming techniques and a wide range of problem domains. The representations and inference techniques should be **general** and **modular**. It should be easy to expand the range of programming techniques that can be identified. We use the game-playing domain as our exemplar, but it should be easy to apply the system to other problem domains with minimal modification.

The remainder of this chapter describes in detail the aspects of the system that realise those objectives.

5.1.2 Overview of the Process

There are two main aspects to the recognition process which correspond to a structural and a behavioural analysis of the student's code. The structural aspect identifies the tasks and techniques in the structure of the code and the behavioural aspect identifies the domain objects and the data structures that the student's code creates. For the sake of clarity of presentation, these aspects are described separately. In the implementation the analysis of these two aspects is interleaved.

Identify tasks and techniques This is done in a sequence of four main phases:

1. **Partition and match the student's code** The student's code is partitioned into recognition nodes. These are groups of predicates in which each group is identified with the realisation of some task. Some predicates may not be identified with any task.

2. **Combine recognition nodes into larger sections** Recognition nodes are combined into recognition graphs, to obtain consistent interpretations for larger sections of the code. If a consistent interpretation cannot be derived, the system comments on the inconsistencies.

3. **Transform unrecognised parts of the program** The system includes a number of transformation operators that can produce variants of the existing code. Application of the operators is guided by the interpretation so far.

4. **Obtain partial information** Some parts of the program may not correspond to any known implementation. The system uses clausal split to identify smaller components that can be recognised and to isolate components that cannot (which are assumed to be bodged).

The end result is an interpretation of the program's structure.

Identify types and domain objects This is done in a sequence of four main phases:

1. **Annotate code with data type information** Type inference is performed on the student's program, and every predicate in the program is annotated with a predicate type declaration. This is done before any other processing, so that data type information can be used by the recognition processes for tasks and techniques.

2. **Obtain types for each domain object used by a task** As recognition nodes are formed for each task, the types for each domain object in the task definition are noted.

3. **Obtain types for domain object** Several tasks may refer to the same domain object. Type information about each domain object is gathered from the task recognition nodes into a set of interpretations for each single domain object.

4. **Identify the components of each domain object** A domain object can only be identified with certainty if all its components can be identified with appropriate domain objects.

The end result is a set of domain structure interpretations for data structures created and manipulated by that program.

5.2 Code Structure: Identify Tasks and Techniques

The objective of our structural analysis is to match the largest possible part of the student's code to the tasks that must be performed in the noughts and crosses problem and to the programming techniques that we have identified. The interpretation of the student's program must be consistent.

A simple top-down strategy for analysis-by-synthesis is to take the top level task, synthesise all possible implementations for that task and find an implementation at the top level in the student's code that matches a synthesised implementation. If any match succeeds, then the tasks corresponding to the sub-tasks at the top level are synthesised and identified in the student's code in turn. This process is be repeated at each level of task and sub-task until all sub-tasks have been identified. We do not use a purely top-down approach to generating code. Code may be matched at a low level even if code at a higher level does not match (e.g. a high-level algorithm is unfamiliar). Instead of generating code top down in the task hierarchy, the system matches each task independently of the tasks that relate to it (e.g. its sub-tasks or super-tasks). Only after all possible tasks have been identified independently in the program as recognition nodes are they connected into a recognition graph.

The analysis starts by trying to identify implementations of different tasks within the code. It pairs a task to the prototypical predicates that might realise that

task, synthesises a realisation from the pair, and tries to match the student's code to the synthesised code in the realisation.

The main features of this process are:

- Code is partitioned into recognisable tasks, each task being implemented by some recognisable collection of programming techniques;
- Code is matched using analysis–by–synthesis, in which code for matching is synthesised from task definitions and prototypical predicates;
- Each task is first recognised independently of other tasks, and then the analyses of tasks and sub–tasks are combined into a recognition graph;
- The recognition graph of tasks and sub–tasks is compared with the call graph of the student's code;
- The recognition graph gives hints about the parts of the student's code that have not been recognised.

In this section, we describe how we partition the user's code into tasks. Each partition contains a single task and a small group of predicates that implement the task. At this stage, sub–tasks of a task are in separate partitions. These partitions are represented by recognition nodes.

Dependencies are computed between predicates that correspond to a task and predicates that correspond to its sub–tasks. For each task that is identified in the user's code we return a (possibly empty) set of its sub–tasks and the corresponding calls in the user code. The system does not, however, check the consistency of all these calls at this stage.

For each predicate in the code, the system tries to find a task whose implementation it matches. We do not want to compare every predicate in detail to every implementation of a task. Instead, the system first checks the student's predicate against the top predicate in a prototype using data type information. If the

check succeeds, code is generated for each task whose implementation uses that prototype. Finally the student's code is matched in detail against the generated code. Code is generated for each task until either it matches the student's code, or else no more tasks remain for that prototype. Generated code is stored in either case, to save generating it repeatedly.

We assume that each task is implemented only once in the student's code. When a task has been identified with some predicates in the student's code, implementations of that task are not matched against any other predicates. We also assume that each of the student's predicates fulfills a single role in the program. When a student's predicate has been identified with the implementation of a given task it cannot consistently be identified with any other task.

In the remainder of this section, we in describe detail the process of code synthesis, code matching and the combination of fragments of the analysis into larger sections. We illustrate this with an analysis of the code in Figure 5–1.

5.2.1 Code Synthesis

Code is synthesised by combining a task description with a prototypical predicate that embodies programming techniques that might be used for that task.

We illustrate the synthesis procedure using one of the examples from Chapter 4 Figure 4–11, in which code to find the position of an empty square is synthesised from the task definition and a prototype.

The synthesis procedure is initiated during the matching process. If the data types of all the arguments to a prototypical predicate match the data types of arguments in a predicate in the student's code, then synthesis is initiated from that prototype. An example of the code to be recognised is shown in Figure 5–1, and the prototype is shown in Figure 5–2. The use of data types is described in Section 5.7.1.

A prototype can be applied to a task if every role slot in the prototype has a matching role description in the task definition. Some sub–tasks in the task

```
find_empty([Sq|Board], 1)   :- is_empty(Sq).
find_empty([Sq|Board], Pos) :- find_empty(Board, P1),
                               Pos is P1 + 1.

is_empty(empty) :- true.
```

This is a very simple implementation which we use for clarity: none of the students used this implementation.

Figure 5–1: Example Code to Find an Empty Square

TESTED_ELEMENT_NAIVE_COUNTER

```
p([H|T], Pos)  :-  T (H),  B (Pos)
p([H|T], Pos)  :-  p(T, P1),  I (P1, Pos)
```

Tested Element with Naive Counter

Figure 5–2: Prototypical Predicate

definition may not correspond to any slot in the prototype. These sub-tasks are not included in the synthesised predicate. This reflects the relationship between tasks and programming techniques: using particular techniques to solve a task may make some sub-tasks unnecessary. No distinction is made between sub-tasks that are required for all implementations and sub-tasks that appear only in some implementations. The relevant parts of the task definition are shown in Figure 5–3.

Each slot in the prototype is paired with the corresponding sub-task in the left

find_empty_square/2

B (Base)	Base = 1
I (I, Next)	Next is I + 1
T (Element)	*empty_square*(Element)

Figure 5–3: Domain Task Definition: Find an empty square

side of the task definition. The slot in the prototype is then replaced by the Prolog code from the right side of the task description for that sub-task.

The substitution is done by unfolding. Each slot in the prototype is treated as a call to a simple Prolog predicate that consists of precisely one clause. The left-hand side of the sub-task would be the head of the called predicate, and the right-hand side, its body. Substituting a sub-task into a slot corresponds to unfolding a called predicate into the body of the calling predicate.

Find the Position of an Empty Square

```
p([H|T], Pos) :- empty_square (H), Pos = 1.
p([H|T], Pos) :- p(T, P1), Pos is P1 + 1.
```

Figure 5–4: Initial Version of Synthesised Prolog Code

Realisation

Task Name: *find_empty_square/2*
Prototype Name: TESTED_ELEMENT_NAIVE_COUNTER
Synthesised Prolog Code: Find the Position of an Empty Square
`p([H|T], 1) :- empty_square (H).`
`p([H|T], Pos) :- p(T, P1), Pos is P1 + 1.`

Functor `p/2` is a place marker. It may match against any predicate and call of arity 2 in a student's code, provided that the name is used everywhere that `p/2` is used in the synthesised predicate.

Functor *empty_square*/1 is a task specification. It may match against any call to a predicate of arity 1 in a student's code, provided that the name is used everywhere that *empty_square*/1 is used in the synthesised predicate. The corresponding predicate will be expected to match a realisation of the *empty_square*/1 task.

Figure 5–5: Synthesised Prolog Code

The resulting code is shown in Figure 5–4. It may include explicit unifications of the form `X = Y` in the body of the clause. In Figure 5–4 the term `Pos = 1` is an example. We assume that these explicit unifications are not likely to appear in a student's code but are an artefact of the synthesis procedure. This code

is tidied by performing these unifications and removing the redundant lines of code that result. This process preserves the meaning of the code in Prolog programs for which the order of execution is not important, although it may change the meaning of programs for which the order in which variables are bound is significant. The process cannot operate if a single variable is unified to two different values in the same clause, as the result of an *or* branch in the clause. In this case the unifications are left unchanged. It is also possible that some code will require an explicit unification, and these unifications are marked in the prototype and left unchanged in the synthesised code. The final code for our example is shown in Figure 5–5, and is stored in a realisation.

Some sub–tasks are not completely specified as Prolog code. Instead, they refer to other tasks which may be implemented in different ways. For example, in Figure 5–3, the sub–task *empty_square* which tests whether a square is empty is defined separately. This task specification appears in the synthesised predicate. The synthesis procedure does not generate predicates for that sub–task. Instead, a note is made of the relationship between the sub–task and the call structure of the student's program. All these relationships are investigated when tasks and sub–tasks are combined. This is described in Section 5.2.3.

5.2.2 Code Matching

When the code has been synthesised, it is compared to the student's predicates. The comparison requires a line–by–line correspondence between the student's code and the synthesised code. Variable names may differ, but each must be used in exactly corresponding positions throughout each clause. Calls to system predicates must be the same. Predicate names (apart from system predicates) may vary but must be used in exactly corresponding positions throughout each clause. Called predicates must have the same arity in the synthesised code as in the student's code. The synthesised code in Figure 5–5 matches the code in Figure 5–1, and a recognition node is created as shown in Figure 5–6.

Synthesised predicates may contain task specifications. These lead to partial recognition nodes that contain links. For example, the synthesised predicate in Figure 5-5 contains a call to another task *empty_square*/1. The matching procedure only succeeds if there is a corresponding call to a predicate in the student's predicate. If the match is successful, a link is created between the task (i.e. *empty_square*/1) and the predicate (i.e. **is_empty/1**) and stored in the recognition node for the task *find_empty_square*/2.

Task: Prototype: User predicate:	*find_empty_square*/2 TESTED_ELEMENT NAIVE_COUNTER **find_empty/2**
Link Task: *empty_square*/1	 User predicate:**is_empty**

Figure 5–6: Example Recognition Node

Some prototypes consist of more than one predicate, connected in a call graph. For example the prototypes with tail recursive counters in Figure 4–10 each consist of two predicates. If the prototype consists of more than one predicate then the called predicates are matched against the appropriate called predicates in the student's code. A successful match for a collection of predicates identifies both the task that is being implemented and the programming techniques that were used to implement that task.

We have used a strict matching strategy and many trivial variants of the synthesised program will fail to match. Other analysis systems use heuristic matching strategies or formal reasoning about program equivalence (Johnson, 1990; Looi, 1988b; Murray, 1988) or match abstractions of the code rather than the code itself (Wills, 1990). These strategies are more flexible than ours and our matching strategy might be replaced with any of them for further experiment. The more flexible the matching, the greater the likelihood that each predicate in the student's program may approximately match the implementations of many different tasks. This leaves a problem of finding the best interpretation. The

trade–off between getting too many matches and too few requires further exploration.

5.2.3 Create Recognition Graphs

The objective of this phase of processing is to combine the parts of the code that have been recognised into larger fragments. Inconsistencies in the analysis are noted and hints are gathered for dealing with unidentified parts of the code.

Some of the tasks that are identified in the student's code will contain sub–tasks that have been identified with calls to other student-defined predicates. If a sub–task has been identified with a call to a predicate, then the called predicate should be identified with the task definition that corresponds to that sub–task. If a predicate is called in different parts of the program, then calls to that predicate should always correspond to calls to the same sub–task.

When a task is identified with some of the student's code, then that constrains how we expect to interpret the remaining code. When the student's code that corresponds to a task calls another predicate that is neither a system predicate nor matched to part of the prototype, then the call should correspond to a sub–task that is defined separately. The called code should also match an implementation of that task. Matches for tasks are first derived independently of one another. When as much of the student's code as possible has been identified with tasks, then these constraints between tasks and sub–tasks are used to confirm that the analysis is consistent and to create expectations about how to interpret parts of the code that have not yet matched any implementation of any task. Using these expectations as a guide, the unmatched parts of the code are transformed and further attempts are made to match them.

In Figure 5–3, the task *find_empty_square* has a separate sub–task *empty_square*, and this appears as a task specification *empty_square/1* in the synthesised code (Figure 5–5). The example predicate `find_empty/2` in Figure 5–1 matches the implementation of this task, and it calls a predicate `is_empty/1` in the same po-

sition as the task specification. This suggests that the user's code for `is_empty/1` should match an implementation of the *empty_square/1* task (not shown).

Three problems may arise. The analysis of a called predicate may have identified it with a different task; or else a called predicate may have been identified with different sub-tasks in different calls; or else the predicate that should match the sub-task matches no known task at all.

In the first two cases, the analysis is inconsistent. The analysis procedure backtracks to generates a further analysis of the program. The analysis procedure backtracks chronologically through the tasks and prototypical predicates that it has identified so far. Analyses are stored as they are created, so that if no consistent analysis is created then the analysis proceeds using one of the inconsistent analyses.

In the third case, the analysis is incomplete. The analysis treats this as a *hint* about the match that is expected between a sub-task and a predicate. Our hints are all top down (although hints could as easily have been created bottom up too). If a task is identified with a piece of code and it calls a sub-task that should correspond to a predicate in the student's code, but that code has not been identified with that sub-task, then the expectation that the predicate should conform to this sub-task is noted as a hint. Hints from code that has been recognised guide the recognition of code that has not. Rather than apply transformations to all unrecognised code, with an overhead in unguided search, the hints guide which transformations should be applied.

5.3 Novel and Erroneous Implementations: Clausal Split

Some parts of the student's code may not correspond to any known programming techniques. In this case we wish to look for recognisable parts of the code at a finer level. This provides supporting evidence that our analysis is correct, and

it allows us to isolate and comment on the parts of the code that cannot be recognised.

In the previous parts of the analysis, we have gathered hints about parts of the code that we have not recognised so far. The hints suggest that a piece of code should match an implementation of a specified task. We therefore consider which transformations we can perform on the unrecognised parts of the code to see if we can confirm what has been recognised. There are a large number of possible transformations, and so we use the hints to direct the search.

5.3.1 Recognising the Components of a Novel Prototypical Procedure

A purely analysis–by–synthesis approach does not work well if one of the components of the student's composed predicate is novel or otherwise cannot be recognised. We have shown in Section 4.5 that a prototypical predicate can be made up by combining smaller prototypes that represent programming techniques. When confronted with code that we cannot recognise, we reverse this procedure to create components which we may be able to recognise.

We wish to work backwards from the student's composed predicate to its components, to identify the techniques in those components for which this is possible and to isolate the other components. In order to isolate these components it is useful to have a *split* operation, the reverse of the join, which splits a joined predicate into its component parts.

We have designed and implemented an algorithm which successfully derives the components of a composed prototype from its components. Given a notion of independent dataflow within the component predicates it decides which auxiliary calls belong in which component.

We now describe the procedure for splitting a composed clause into its components. The inputs to the process are the composed predicate and the join specification. The program that is used as an example is shown in Appendix C.

5.3.2 The Join Specification

Like a clausal join, a clausal split is controlled by a join specification. The join specification maps between arguments in the composed predicate and arguments in its component parts. The general form of a join specification is shown in Figure 5–7. The join specification does not indicate which auxiliary calls belong in which

$$joined(A_1, \ldots, Aj) \Leftarrow split_1(A_p \ldots A_q) \bowtie \ldots, split_m(A_r, \ldots, A_s)$$

Each set of arguments on the RHS of the join specification (e.g. $A_p \ldots A_q$) is a proper subset of the arguments on the LHS $(A_1, \ldots, Aj)$. All of the arguments on the LHS are represented at least once on the RHS.

Figure 5–7: Form of a Join Specification

$$count\_and\_sum(List, Count, Sum) \Leftarrow count(List, Count) \bowtie sum(List, Sum)$$

Figure 5–8: Example of a Join Specification

component. This problem does not arise in the join, in which all auxiliary calls are simply transferred from the components into the composed predicate. We can use data flow between variables to decide where each auxiliary call belongs.

5.3.3 The Clausal Split Algorithm

The algorithm proceeds one clause at a time through the composed predicate. It splits each clause of the composed predicate into one corresponding clause for each of the components. Each composed clause is split independently, and the matches between variables that are created in splitting one composed clause do not affect the matches for other composed clauses.

We illustrate the process by splitting the second clause of the predicate `count_and_sum/3` in Figure 5–9 using the join specification in Figure 5–8, to give the result in Figure 5–10. The predicate `count_and_sum/3` counts the elements in a list of numbers and also computes their sum. It is to be split into

Count and compute the sum of the elements in a list

```
count_and_sum([], 0, 0).
count_and_sum([H|T], Count, Sum) :- count_and_sum(T, C1, S1),
                                    Sum is S1 + H,
                                    Count is C1 + 1.
```

Figure 5–9: The Composed Predicate

Count the elements in a list

```
count([], 0).
count([H|T], Count) :- count(T, C1),
                       Count is C1 + 1.
```

Compute the sum of the elements in a list

```
sum([], 0).
sum([H|T], Sum) :- sum(T, S1),
                   Sum is S1 + H.
```

Figure 5–10: The Component Predicates

two components, one of which `count/2` counts the elements and the other `sum/2` computes the sum. If both tasks are needed, say to compute the average of a list of numbers, then it is more efficient to call the composed predicate than it would be call the two components one after the other. In the composed predicate the list is traversed only once, whereas it must be traversed once for each component. This example has been chosen even though it is not part of the noughts and crosses task because it illustrates various aspects of the algorithm, in particular:

- variables that are required in both components;
- variables that are only required in one component;
- recursion;
- calls to auxiliary predicates.

Obtain the heads of the split clauses A copy of the join specification is taken and the LHS of the new join specification is unified with the head of the composite clause. This transfers the corresponding arguments to the RHS of the new join specification.

```
count_and_sum([H|T], Count, Sum) ⇐
        count([H|T], Count) ⋈ sum([H|T], Sum)
```

The resulting terms in the RHS are the heads of all the components, as below:

```
count([H|T], Count) :- ....
sum([H|T], Sum) :- .....
```

All variables that appear in more than one call on the RHS of the join specification are entered into the set of *shared* variables and all variables that appear in only one call on the RHS of the join specification are entered into the set of *private* variables for that call. In our example, the sets are:

Private to count:	Count
Private to sum:	Sum
Shared:	H, T

Deal with recursive calls If there are any recursive calls in the body of the composite clause, each recursive call is split in the same way as the heads. To ensure that variables are bound correctly, a fresh copy of the join specification is taken for each recursive call in the composite clause. The composite call is unified with the LHS of the join specification and the arguments are transferred to the RHS.

```
count_and_sum(T, C1, S1) ⇐
        count(T, C1) ⋈ sum(T, S1)
```

The recursive calls are placed in the bodies of the split clauses. The unifications between the copies of the join specification and the composed predicates have ensured that the right arguments appear in both the heads and the recursive calls, so no further work is needed to do this.

```
count([H|T], Count) :- ... count(T, C1) ...
sum([H|T], Sum) :-  ... sum(T, S1) ...
```

The sets of private variables are extended so that any variable that is included in a split recursive call but is not a shared variable, is assumed to be private to that split clause and is added to the set of private variables for that clause. In our example, the sets are:

```
Private to count: Count, C1
Private to sum:   Sum, S1
Shared:           H, T
```

Decide which auxiliary calls are to be included By this stage, one clause has been created for each component predicate, with an appropriate head and appropriate recursive calls. Sets have been created of the variables known so far to be private to each component, and also a set of variables that are shared between components. The algorithm now decides which auxiliary calls should be included in which components.

Repeated passes are made through the auxiliary calls in the composite clause. If a call refers to any variables that are private to a component, then the call is added to the appropriate component predicate, and any new variables in the call are added to the set of private variables for that component. There is also a check that none of the other variables in the call are private to any other component, as this would mean that a correct split is not possible. This procedure is repeated until there is a pass for which no more auxiliary calls have been added to any component predicate.

Any auxiliary calls that remain at the end of this process do not contain any private variables. We assume that such calls are relevant to all the components, since they do not include any dataflow that is private to a single component, and so they are copied into every component predicate.

The predicate `count_and_sum/3` contains two auxiliary calls. The call `Count is C1+1` refers only to variables private to `count/2`, and is therefore included in the body of `count/2`. The call `Sum=S1+H` refers to variables private to

sum/2 and to an explicitly shared variable and is therefore included in the body of sum. No further extensions are made to the sets of private variables. The resulting clauses are:

```
count([H|T], Count)  :-  count(T, C1),
                         Count is C1 + 1.

    sum([H|T], Sum)  :-  sum(T, S1),
                         Sum is S1 + 1.
```

Continue This process is repeated for each remaining clause in the composed predicate. The first clause of the predicate count_and_sum/3 in Figure 5-9 has no body, and so it requires only the first part of the algorithm to obtain the heads of the split clauses. The end result is the predicates in Figure 5-10.

5.3.4 Dataflow and the Join Specification

We cannot split predicates which contain dataflow dependencies between variables unless those dependencies have been captured by the join specification. This means that, as for clausal join, all variables on the RHS of the join specification must appear on the LHS. It also means that if the join specification indicates that two variables belong in two different components but a single auxiliary call refers to both variables, then we cannot tell what to do with the auxiliary call and the split cannot proceed. A (somewhat contrived) example in which a clause could not be split would be if the second clause of count_and_sum/3 contained an extra call RunningTotal is Sum + Count, perhaps in order to display this running total:

```
    count_and_sum([H|T], Count, Sum) :-
        count_and_sum(T, C1, S1),
        Sum is S1 + H,
        Count is C1 + 1,
        RunningTotal is Sum + Count,
display_so_far(RunningTotal).
```

`Sum` is private to one component, `Count` to the other. The auxiliary call `RunningTotal is Sum + Count` refers to both variables and so it cannot be assigned to either component. This clause could not be split.

If an auxiliary call cannot be assigned to one of the components and if there exists a join specification for the auxiliary call, then we recursively split both the auxiliary call and the predicate that is called. Hence a prototype which creates more than one predicate has a join specification for each predicate, so that all the predicates for that prototype may be split.

Strictly speaking, where a join specification joins more than two components we may need to distinguish between variables that are shared between all the components, and variables which are shared between different subsets of components. This is a straightforward extension which we do not discuss further.

5.3.5 The Application of Clausal Split

Our objective in using clausal split is to help the system to identify recognisable components of a student's predicate even when the predicate as a whole cannot be identified. If the system has formed a hint that the student's code is intended to perform a particular task, then it may use the split to obtain supporting evidence that the student did indeed intend to implement that task. The split also enables the system to identify some of the programming techniques that the student has used in the code and to hypothesise the purpose of the components that it cannot recognise.

The system must decide which join specification is appropriate. In general, the number of possible join specifications that could be tried for any one predicate is large. It is likely that only a small proportion will result in successful splits, and only a small proportion of these might result in splits that can be recognised. We would like therefore to constrain the system's search to splits that are likely to be successful and lead to recognisable components.

We restrict our application of clausal split for use only when the role slots in the predicate are not split, i.e. each role slot falls into one of the component predicates. In this case, performing a clausal split on a prototype and then using the component prototypes to realise a task will synthesise the same code as using a prototype to realise a task and then performing a clausal split on the synthesised predicates. The knowledge base of prototypical predicates contains join descriptions that give explicit information about which prototypes create these components and the join specifications that would be needed to create them from the composed code. Each prototype that can be split contains a join specification for each predicate within the composed prototype and the names of the component prototypes that are expected. (Figure 5–12 gives an example of the join information that is associated with a prototype.) When code that has been synthesised for a task from the composed predicate is split into components using the associated join specifications for that prototype, the components correspond to code that has been synthesised for the same task from the named component prototypes.

A clausal split is applied when there is a hint that one of the student's predicates should correspond to a particular task, but the code does not match any implementation of that task that we can synthesise. The analysis procedure looks at the knowledge base to see if any of the synthesised implementations for that task has used a prototypical predicate which is the result of a composition (i.e. it has join specifications). If so, then the join specifications are retrieved, and the student's code is split according to that specification. Code for comparison is generated from the expected task and from the component prototypes that are specified with the join specification. The components of the student's code are compared with the synthesised components. If at least one component matches, then we take this to confirm the student's intention to implement that task, and we comment on the parts of the code that do not match and are therefore not a standard technique for that task.

We use clausal split to identify the components of an implementation that might

correspond to a recognisable part of a prototypical predicate. When the components have been separated, they may have different control behaviour from each other and from the combined code. All of these components are part of the same task and they do not correspond to independent sub-tasks. Instead the components represent smaller groups of programming techniques that contribute to the implementation of the task.

This contrasts with Sterling and Lakhotia's use of clausal join for software synthesis (Sterling & Lakhotia, 1988). They use clausal join in order to combine pieces of code which performs a sequence of different tasks using the same control into a single piece of code which performs all the tasks in parallel. As a result, they require that each component should have the same control flow (in their terminology, all components to a join should arise from the same control *skeleton*) and no component should have altered the control flow of the skeleton. This constraint ensures that the composed code also has a similar control structure to its components. The inverse application of Sterling and Lakhotia's procedure would create independent implementations of the different tasks, each with similar control to the combined code. We require that components have independent dataflow other than what is captured by the join expression, but we do not require that components have identical control to each other or to the composed code.

5.3.6 Using Clausal Split for Partial Recognition

This section presents an example of using clausal split to recognise a bodged implementation. Figure 5–11 presents one student's implementation of the predicates that find the position of the next empty square[1]. The implementation works, but it cannot be matched to any synthesised implementation finding an empty square because the student has not implemented a counter properly.

---

[1]Taken from program P12 with predicate and variable names altered

Student P12's Code to Find the Position of an Empty Square

```
next_empty_square(L, C) :-
     next_empty_square_sub(L, [1,2,3,4,5,6,7,8,9], C).
next_empty_square_sub([H|_], [C|_], C) :-
     is_empty(H).
next_empty_square_sub([_|T], [_|Cs], C) :-
     next_empty_square_sub(T, Cs, C).
```

Figure 5–11: Example of Bodged Prolog Code

The analysis of the code that calls the predicate **next_empty_square/2** creates a link which suggests that this code should perform the task of finding an empty square, but it matches none of the synthesised implementations. Four of the prototypes that can be used to synthesise code for this task have join specifications that divide the prototype into two different components. These prototypes and join specifications are shown in Chapter 4, Figure 4–10. In each case the components are a counter (Figure 4–6) and a list recursion (Figure 4–5).

Tested Element with Tail Recursive Counter

Prototype:

```
      p(List, Pos)  :-  B (Base),
                        q(List, Base, Pos)
q([H|T], Pos, Pos)  :-  T (H)
  q([H|T], P, Pos)  :-  I (P, Next),
                        q(T, Next, Pos)
```

join specifications:

$$position_1(List, Count) \Leftarrow test_1(List) \bowtie count_1(Count)$$
$$position_2(List, Acc, Count) \Leftarrow test_2(List) \bowtie count_2(Acc, Count)$$

Expected Component Prototypes:

$test_1, test_2$	Expanded version of Tested Element Prototype
$count_1, count_2$	Tail Recursive Counter Prototype

Figure 5–12: The Composed Prototype with Join Information

Each split is tried in turn until a successful split and a partial match are found. We describe only the successful split, using the prototype and join specifications

in Figure 5–12 and the clausal split procedure described in Section 5.3. Splitting the bodged code in Figure 5–11 results in the code shown in Figure 5–13.

```
test_1(L) :- test_2(L).
test_2([H|_]) :- is_empty(H).
test_2([_|T]) :- test_2(T).

counter_1(C) :- counter_2([1,2,3,4,5,6,7,8,9],C).
counter_2([C|_], C).
counter_2([_|Cs], C) :- counter_2(Cs, C).
```

Figure 5–13: The Results of Splitting the Bodged Code

The component prototypes that we would expect to identify from a correct implementation are shown in Figure 5–14 and the code that would be obtained from each prototype is shown in Figure 5–15.

The information about components in Figure 5–12 indicates how to match this code. This information indicates that the prototypical predicates that we expect to find consist of a variant of a list test plus a tail recursive counter. These prototypes are either looked up or else created. The list test prototype and the tail recursive counter prototype are pre-stored, but the variant of the list test prototype must be created. The variant is created by wrapping the original recursive list test as shown in Figure 4–5 in an extra predicate, giving the version in Figure 5–14.

Code is then synthesised for each component of the task using the task description plus the component prototypes. This code is shown in Figure 5–15. The synthesised code is compared with the two components of the student's code. One

Expanded version of Tested Element	Tail Recursive Counter
`p(L) :- q(L)`	`p(C) :- B (B), q(B, C)`
`q([H|T]) :- T (H)`	`q(C, C) :- true`
`q([H|T]) :- q(T)`	`q(C1, C) :- I (C1, C2), q(C2, C)`

Figure 5–14: The Associated Component Prototypes

Finding an Empty Square

Expanded version of Tested Element	Tail Recursive Counter

`p(L) : q(L).`
`q([H|T]) :-` *empty_square* `(H).`
`q([H|T]) :- q(T).`
	`p(C) :- q(1, C).`
	`q(C, C) :- true.`
	`q(C1, C) :- C2 is C1 + 1,`
	`q(C2, C).`

Figure 5–15: Code Synthesised from the Component Prototypes

component, `test_1/1` and `test_2/1`, matches the code synthesised from the extended recursive test. The second component `counter_1/1` and `counter_2/2` does not match the code synthesised from the tail recursive counter. This result supports the hypothesis that the student's code is indeed an implementation of finding an empty square. It also suggests that the recursive list testing component has been implemented correctly but the counter has been implemented by a novel or incorrect method.

5.4 Dealing with Code Variants

Ideally, the system may generate variants of the code that it has synthesised, using transformation operators which preserve the meaning of the code. If the transformed code corresponds to the student's code, then the system has recognised the overall tasks and the programming techniques being used. We have implemented some transformation operators, which we describe here for completeness, but we have not incorporated them into the analysis.

We have implemented the following transformation operators:

unfold a called predicate into the body of a caller;

divide an "or" branch in the body of a clause into separate clauses;

clausal join predicates with similar control structure into a single predicate;

clausal split a predicate into two predicates with a similar control structure.

```
find_empty([empty|Board], 1) :- true.
find_empty([Sq|Board], Pos)  :- find_empty(Board, P1),
                                Pos is P1 + 1.
```

This is a variant of Figure 5–1. In this variant the predicate `is_empty/2` which tests an empty square has been unfolded into the body of the first clause and replaced by a pattern match in the head of the clause. The sub–task `empty_square/1` no longer corresponds to a separate predicate.

Figure 5–16: Variant Code to Find an Empty Square

The unfolding of calls is important to matching because the synthesised code will have a particular call structure as a result of the task structure, and this call structure may not correspond to the student's structure. In our architecture, calls to an explicit sub–task may be used to express one of two distinct possibilities.

The first use for a distinct sub–task is to express a sub–task that is conceptually distinct from its super–task and therefore that same sub–task might be used by other tasks. We expect, as a principle of good design, that distinct tasks should be implemented by different predicates. For this reason, a call to a named sub–task is always initially synthesised as a call to a separate predicate.

The second use for a distinct sub–task is to express an *or*–branch in the task structure. It is possible to describe different approaches to a problem in a language–independent way by breaking down the problem into different collections of sub–tasks. In this case, sub–tasks are not necessarily called from elsewhere and they may be very simple, in which case they may reasonably be incorporated into the body of the calling predicate.

We need to deal with cases when the student has incorporated the code for a distinct sub–task into the body of the predicate for the super–task.

If a sub–task is called only in one part of the code and if it is very simple, then code for the predicate may reasonably be unfolded into the body of the predicate. In other cases, complex tasks may have been so intertwined that the resulting code obscures the task structure, and this can be seen as a design flaw in the

code. An example is shown in Figure 5-16, in which the the call to `is_empty/1` has been unfolded into the body of `find_empty/2`.

Our automated analysis does not address the question of when we would consider such transformations as legitimate and when they should be considered to be errors (or mal-rules (van Someren, 1990)). Whether or not we consider that in any particular case the combined code is flawed, we still need to recognise code in which this has happened. From a systems engineering point of view, such transformations are vital to cut down the number of task descriptions that must be explicitly represented.

5.5 Assess the Effectiveness of the Analysis

Each program may result in different interpretations. To begin with, as many tasks as possible are identified within the student's program. Some of the student's predicates could be interpreted in different ways as belonging to different tasks. In each interpretation of the program, each predicate is identified with at most one task.

The reporting procedure identifies the complete and incomplete tasks. A complete task is one all of whose sub-tasks have been completely recognised in the code. An incomplete task is one with at least one sub-task at some level that has not been identified. Such a call to an unidentified sub-task is a *hole*.

The reporting procedure computes a size for each task. This is defined recursively to be one for a task plus the sum of the sizes of each sub-task within the task that has been identified with the user's code. The reporting procedure also counts the number of holes in each incomplete task, defined recursively to be one for each unidentified sub-task within the task plus the sum of the number of holes within each identified sub-task.

Find an Empty Square

$\mathcal{B}$ (Base)	`Base = 1`
$\mathcal{I}$ (I, Next)	`Next is I + 1`
$\mathcal{T}$ (Element)	`empty_square(Element)`

Domain objects: BOARD
PLACE

Figure 5–17: Domain Objects in Task Definition

The name and size of each task that has been completely recognised is reported to the user. The name, size and number of unidentified sub–tasks are reported for incomplete tasks.

The number, size and coherence of the tasks recognised forms a measure of how good the match really is. Each interpretation is given a quality score. The system counts the number of predicates that form part of completely and incompletely recognised tasks. Each of these is computed as a proportion of the number of predicates in the student's code. This quality measure gives less weight to incomplete tasks by adding only half the proportion of predicates from incomplete tasks to the full proportion of predicates from complete tasks.

5.6 Code Behaviour: Describe Domain Objects

Each domain object that may appear in the game is specified by a collection of domain object descriptions. Each description specifies one way in which domain objects of that kind (e.g. boards, or lines, or squares) may be represented. Each domain object description consists of a name plus a *type constraint* plus zero or more *part constraints.*

Task descriptions contain information about the domain objects that are represented by their arguments. An example is shown in Figure 5–17. The domain

objects in the task definition are ordered, and this order corresponds to the order of arguments in all the predicates that are synthesised for that task. If an implementation of that task consists of more than one predicate, then the domain objects correspond to the arguments to the top predicate. When the code in Figure 5–5 is synthesised from the task definition in Figure 5–17, this means that the first argument in that code represents the BOARD and the second represents the PLACE where an empty square has been found.

Data types are inferred for every argument to each predicate that the student has defined by the process described in Section 5.7. When a student's predicate is identified with a task, the inferred data types for the arguments of the predicate are also identified with the appropriate domain objects in the task. As tasks are identified, the data types of domain objects manipulated by those tasks are reported to the user. The system notes the data type of the domain object together with the match between task and predicate that has led to the inference.

At first, these notes about domain objects are independent: matches for other tasks may lead to different or even contradictory inferences about domain objects. Just as the consistency of tasks is checked when individual tasks are gathered into groups, so the information about each domain object is also gathered together and checked for consistency. Data types for the same object are unified, so that less specific information about data types can be combined with more specific information. At the end of the analysis this information is reported to the user (Figure 5–18).

5.6.1 Store Annotated Code

The student's code is read and stored clause by clause. Type inference is performed on the student's code and the data types of the arguments for each predicate in the student's code are stored. The type inference mechanism is described in Section 5.7.

Domain object BOARD is given type *list(atom_type)* by:

Predicates	Tasks:
`find_line/2`	*find_line/2*
`choose_move/3`	*choose_move/2*
`fill_a_pair/3`	*line_next_move/2*
`next_empty_square/2`	*find_empty_square/2*

Domain object BOARD is described by *list_of_squares*

Domain object SQUARE is given type *atom_type* by:

Predicates:	Tasks:
`empty_square/1`	*empty_square/1*

Domain object BOARD is described by *atomic_square*

Domain object PLACE is given type *integer* by:

Predicates:	Tasks:
`fill_a_pair/3`	*line_next_move/2*
`next_empty_square/2`	*find_empty_square/2*

Domain object PLACE is described by *numerical_id*

Figure 5–18: Results of Data Analysis for Three Domain Objects

5.6.2 Identify Domain Object Types in Tasks

Some tasks are sufficiently specific to be performed only on a single domain object per argument. The definitions for each of these tasks contain domain object descriptors to specify which domain object is represented by each argument in an implementation of the task. When a student's code is matched to a realisation of a task, the domain objects in the task are identified with the arguments in the student's predicate. If the realisation for a task consists of more than one predicate, then it is the arguments to the top predicate in the call graph that correspond to the domain objects in the task.

A predicate type declaration has already been inferred for every predicate in the student's program. For example, given the code in Figure 5–1, the types of the student's predicate `find_empty_square/2` are *list(atom_type)* and *integer*. When a domain object is identified with an argument in the student's predicate, then the data type of that argument in the predicate type declaration corresponds

to the type that the student has used to represent that object in that task. As shown in Figure 5–17 the corresponding domain objects are BOARD and PLACE.

At this stage, data type information about domain objects is stored in the recognition node for that task (Figure 5–19).

Task: Prototype: User predicate:	*find_empty_square/2* TESTED_ELEMENT NAIVE_COUNTER **find_empty/2**
Link Task: *empty_square/1*	 User predicate:**is_empty**
Domain objects: BOARD PLACE	Data Types *list(atom_type)* *integer*

Figure 5–19: Example Recognition Node

5.6.3 Identify Domain Object Representations

Different representations for the same domain object may be more suitable for different tasks, so we consider that it is acceptable for a domain object to be represented in different ways in different procedures. As a consequence, we expect to collect not just one data type for each domain object but a set of different data types for each domain object (possibly an empty or unitary set) and we expect to infer a different representation for the domain object from each data type in the set. With each data representation in the set we note where it came from, i.e. the user's procedures and the matching tasks that supplied the type and data representation inference. All of the predicates in Figure 5–18 use the same representation for each domain object.

When the recognition graphs have been explored to create an interpretation for the code, the domain data structures are also combined to create an interpretation for the data structures. The instances of a domain object from different tasks are grouped together according to whether they have the same data type. Each group is a potential interpretation for that domain object. More general

types for a domain object (i.e. types that contain type variables) are also combined with more specific types by unification. At this stage each domain object interpretation contains a type (that cannot be unified with the type for any other interpretation of this domain object) and the names of tasks and student predicates in which that domain object was assigned that type (or a more general version of it).

Finally, the system works out what the type information implies about the representation for domain objects. For each interpretation, the data type is matched to the domain object definition. The appropriate domain object definitions are shown in Figure 4–18. The data type is unified with the type constraint. If there is no part constraint, then a successful unification means that the domain object has been successfully identified and the representation for the domain object consists of the name of that domain object description. In the example task *find_empty_square*, the domain object PLACE has been identified to have type *integer* in the student's code. This satisfies the type constraint for PLACE, and since it has no part constraints the system infers that its representation is *numeric_id*.

If there are part constraints, these part constraints must also be satisfied. The unification between the data type and the type constraint has the effect of binding a variable in the part constraint to a type, the type of the part to be checked. The inference process recursively checks that the contents of part constraints correspond to one of the possible representations for the specified domain object. Thus in the example task *find_empty_square*, the domain object BOARD has been identified to have type *list(atom_type)*. This can unify with the type constraint for two possible representations, *list_of_lines* or *list_of_squares*, depending on whether the type *atom_type* corresponds to a LINE or to a SQUARE.

The type of the part is checked in two ways. First, the type is checked against the type constraint for that part. In our example, the type *atom_type* unifies with one of the type constraints for a SQUARE but not for a LINE. This implies that the board should be interpreted as a *list_of_squares*. Second, a further check is

performed. The system looks for any data interpretation in which a SQUARE has been identified in the student's program as having the type *atom_type*. Assuming that the student's predicate `is_empty/1` has an argument of type *atom_type* and that this predicate has been identified with the task *empty_square* which has the domain object SQUARE, then the match is successful and the representation of the BOARD is indeed a *list_of_lines*.

The second check makes the success of the domain object recognition depend heavily on the success of the task recognition. Parts of a domain object are recognised if a task that uses that part is also recognised, and the whole domain object is only recognised only if all its parts are recognised. The second check was introduced to reduce the potential for multiple inappropriate interpretations for a domain object. The task recognition is quite fragile and so the second check is unnecessarily severe and tends to hamper recognition. It would be sufficient to perform only the first check, and it would be helpful to the recognition if this domain object inference were used to drive the recognition of tasks that use the parts of the domain objects.

The results of this analysis for three domain objects in a program that contains several tasks are shown in Figure 5-18.

5.7 The Type Inference Mechanism

Polymorphic type inference provides a mechanism by which the system can describe and identify the data structures that are used and created by predicates. We use it to describe the behaviour of Prolog programs and the intentions of students in writing those programs.

The type inference mechanism works as follows. To infer the argument types of a predicate, the type inference mechanism first creates an empty structure that contains a type variable for each argument to the predicate. Then it looks to see if a predicate type declaration exists for that predicate. If such a declaration

Data Type Declarations:

list(*T*)	←	[] ∪ [*T* | *list*(*T*)]
square	←	x ∪ o ∪ empty
integer	←	*integer* + *integer*

Initial Predicate Type Declaration:

```
is(integer, integer)
```

Input Prolog Program:

```
find_empty_square([empty|_T], 1).
find_empty_square([_H|Rest], Position) :-
        find_empty_square(Rest, P1),
        Position is P1+1.
```

Predicate Type Declarations After Type Inference:

```
is(integer, integer)
find_empty_square(list(square), integer)
```

Figure 5–20: Illustrative Example of Type Inference

already exists, the types in the predicate type declaration are bound to the arguments in the type structure and the process terminates. If it does not exist, then the types of the arguments are inferred from the predicate itself. In this case the type inference mechanism processes each clause in turn.

To infer the data type declaration for the second clause for the predicate `find_empty_square/2` in Figure 5–20, a structure is created with one type variable for each argument in the head of the clause:

$$T_1, T_2$$

First, the head of the clause is processed. Each argument is processed in turn. If the argument is a variable, then the variable is stored in a bindings list and in the type structure. If the argument is not a variable, it is matched (by unification) to the RHS of some data type declaration. The corresponding type is stored in the type structure. An argument that is not a variable itself may contain variables. These variables are stored in the bindings list. An argument may be a complex term, in which case each part of the term is either a term that is

matched recursively to the RHS of a type declaration and its type is returned, or else it is a variable that is added to the bindings list.

In the example, the first argument is `[_H|Rest]` and the second argument is `Position`. The first argument unifies with the RHS of the data type declaration for a *list(T)*, and the second is just a variable. The type structure is now:

$$list(T_3), T_2$$

and the bindings are

`_H`	T_3
`Rest`	$list(T_3)$
`Position`	T_2

The bindings list contains exactly one entry for each variable in the clause (no matter how many times it appears in the clause) and its type. Bindings are maintained to appropriate types or type variables for each argument in the head of the clause and for each variable referenced in the head or body of the clause. Each time type information is derived for a variable, that information is combined with the type information for that variable in the bindings list by unification.

Next, types are derived for variables in all calls in the body of the clause. The body of the clause is assumed to consist of a series of calls connected by *and* and *or* connectives, i.e. `','` and `';'` in Prolog syntax. The body of the clause is parsed and each call is dealt with in turn. For each call, the calling type is derived (as for the head). New variables are added to the bindings list. Types for variables that have already been identified are looked up in the bindings list.

The first call in the clause is `find_empty_square(Rest, P1)`. This adds one entry to the bindings list, which is now:

`_H`	T_3
`Rest`	$list(T_3)$
`Position`	T_2
`P1`	T_4

The predicate type declaration of the called predicate is found, either by looking it up an existing declaration (e.g. the declaration for `is/2`) or else by a recursive call to the type inference mechanism on the called predicate. In the case where the predicate call is itself recursive, as is the first call in the second clause of `find_empty_square/2` in Figure 5–20, a stack is maintained of the type information for the predicates that have been analysed so far. This information is restricted to the current clause in the current predicate, not the whole predicate, which is a weakness in our mechanism but simplifies the processing.

The types obtained for each argument of the call are compared (by unification) with the types that are obtained for each argument to the called predicate. The comparison is done by unifying the types of the arguments, which may lead to further instantiation of type variables in the variable bindings list. In our example, the recursive call adds no new information, since it is already known that `Rest` is of type $list(T)$ and the type of `P1` is unconstrained. However, this does act as a confirmation that the recursion is using types as expected.

This process is repeated for each call in the clause. Thus the call `Position is P1 + 1` is handled similarly. The first arguments `Position` is a variable which is already on the bindings list. The second argument `P1 + 1` is looked up as a data type whose arguments are expected to be *integer* leading to an *integer* result. This adds the information to the bindings list that `P1` is of type *integer*. Looking up the predicate type definition of `is/2` confirms that `is/2` has been used correctly on integer arguments. It also adds the information that the type of `Position` is *integer*. At this stage, the variable bindings are:

`_H`	T_3
`Rest`	$list(T_3)$
`Position`	T_2, *integer*
`P1`	T_4, *integer*

The binding for `Position` now states that T_2 is *integer*, so the predicate type for this clause is now:

$list(T_3), integer$

The process is repeated for the other clauses. By the same process, the first clause `find_empty_square([empty|_], 1)` derives the following information for its arguments:

$list(square), integer$

When types have been derived for the arguments to each clause, the types for each argument are combined from all the clauses by unification, to give the overall type of the predicate. In our example, the resulting predicate type declaration is:

`find_empty_square`$(list(square), integer)$

An extra mechanism must be supplied to deal with numbers, and with any other types with an infinite number of data constructors whose values therefore cannot be enumerated explicitly. This is easily done for types like *integer* which are built in to Prolog. Instead of comparing the value with the RHS of a data type declaration, the value is tested by calling the built-in Prolog predicate `integer/1`. If the test succeeds then the variable is of type *integer*.

Figure 5–20 is necessarily only a simple example to illustrate the inference mechanism. Our system is able to infer types for student programs whose predicates are deeply nested, recursive and mutually recursive.

5.7.1 Using Data Types to Constrain Search

Type information is used to cut down the amount of detailed code matching. If the student's predicate manipulates quite different data types from the prototype then the code of the student's predicate will not match the code that is generated

```
p([H|T], Pos)   :-  T (H),  B (Pos)
p([H|T], Pos)   :-  p(T, X, P1),  I (P1, Pos)

    Data types  :   p(list(L), P)
```

Figure 5–21: Prototypical Predicate for Position of a Tested Element

```
  find_empty([Sq|Board], 1)  :-  is_empty(Sq)
find_empty([Sq|Board], Pos)  :-  find_empty(Board, P1),
                                 Pos is P1 + 1

          is_empty(empty)    :-  true

              Data types     :   find_empty(list(atom_type), integer)
                             :   is_empty(atom_type)
```

Figure 5–22: Student's Code to Find an Empty Square

The data type information for the prototypical predicate is less detailed than the type information for the student's code, but they are compatible. The types and type variables for the the prototype can unify with the types for the student's code.

from the prototype. The predicates in the student's code are fully specified and they call other predicates, so their data types are inferred in detail. The prototypes are not so fully specified. Prototypical code has yet to be completed by tasks and we do not know all of the other predicates that it might call, and so we have less detailed information about its data types. We do not therefore insist on an exact match, but we accept as successful a match in which the data types of the arguments to the prototype unify with the data types of arguments to the student's predicates (e.g. Figures 5–21 and 5–22).

5.7.2 Scope and Limitations of the Type Inference Mechanism

Our mechanism cannot obtain any type information from predicates which take executable code as data input and execute that code. There are two main ways in which Prolog supports this. Firstly, Prolog supports the built-in predicates

`setof` and `bagof`, both of which take as input a procedure to be executed and return a list of values arising from that execution. Mycroft and O'Keefe's type inference mechanism is able to support this (Mycroft & O'Keefe, 1984). Some students did use these predicates. We are able to derive the information that they return a list, but no further information. This would be a straightforward extension to our type inference mechanism. The second feature that Prolog supports is more general. The system predicates `univ` and `call` respectively create and run executable code. It is not possible to obtain type information about the results of calls to procedures that are only created at run-time, and Mycroft and O'Keefe's type inference mechanism does not support this. None of the students we studied used this mechanism for this exercise, and it is considered an advanced programming technique.

5.8 Novel and Erroneous Data Implementations

5.8.1 Anonymous Types

In trying to apply type inference to the understanding of student programs, we encountered a search problem entailed by trying to derive detailed data type information from arbitrary programs. We have resolved this problem by allowing type inference at different levels of detail within the same system, as follows. The type language allows type declarations that are based on the functor and arity of the data constructors as described in Section 4.12. The language also allows less detailed anonymous types, as described in Section 4.12.2.

If the inference mechanism cannot infer a type for some data structure using the specific type declarations that have been built in for particular functors and arities, then the mechanism tries to infer an alternative data type using the anonymous types that are based only on the arity of the the data structure. The same type inference mechanism is used in both cases.

An anonymous data type can be derived in this way for an argument to a predicate that uses different functors, so long as all the functors have the same arity. If a single argument is inferred to pass data structures whose functors have different arities then no data type information can be inferred.

5.8.2 Dealing with Inconsistencies

Type inconsistencies may be detected at various points. They may be found where a predicate is called with an argument of some incompatible type, where a call unifies two variables which have different types, or where different clauses in the same predicate have arguments of incompatible types.

We have implemented a flexible mechanism which can treat inconsistencies with different degrees of stringency. In a type checker whose purpose is to check that types have been used consistently, a predicate is treated as inconsistent if it calls a predicate whose types are inconsistent. Mycroft and O'Keefe's type checker derives no type information for the calling predicate, merely stopping with an error message at the first inconsistency it finds (Mycroft & O'Keefe, 1984). By contrast, our mechanism derives plausible information about intended types, and so type information can be obtained for a predicate even if it calls a predicate whose types are inconsistent. The call to the inconsistently typed predicate simply supplies no information to the types that are inferred for the caller. Instead of flagging an inconsistency, the call returns open type variables so that the inference may proceed with the calling predicate. This allows the mechanism to derive some plausible information about the intended data types of the called predicate and some limited information about the data types of the caller.

5.9 Summary

This chapter provides an overview of the procedure by which the system analyses a student's program. It then describes in more detail the way in which the parts of the analysis procedure have been implemented to identify code structures and data objects.

The following aspects of code structure analysis were described in detail:

- the synthesis of code from task specifications and prototypical predicates;
- the identification of the student's code structure with problem-specific tasks and language-specific programming techniques using analysis-by-synthesis;
- the combination of small sections of the analysis into larger consistent sections, and the way in which the analysis so far is assessed;
- the hints mechanism and the recognition of unfamiliar code using clausal split.

The following aspects of the analysis of code behaviour were described in detail:

- the definition of domain objects;
- the type inference mechanism;
- the combination of the two to recognise decisions about data representation.

Chapter 6

Results: An Analysis of Example Programs

6.1 Introduction

In this chapter we describe the results of running our analysis software on a range of programs. We demonstrate the behaviour of the program and we evaluate the system's ability to recognise design decisions. The demonstrations in this chapter assess the ease with which programs may be understood and design decisions recognised, including programs that are novel or bodged.

The elements of recognising design decisions are closely entwined. We can derive information about decisions that are independent of the problem domain, but to critique them effectively we need information about the student's intentions in terms of the problem domain.

We used the following demonstrations to assess our system:

6.1.1 Demonstration 1: Idealised programs

We consider an example implementation of the core of a noughts and crosses program that runs correctly and follows the specification as given.

This demonstration illustrates:

- The recognition of individual tasks and techniques;
- The recognition of domain objects and the data structures used to implement them;
- The combination of fragments of the analysis into larger fragments.

The demonstration programs consist of the core parts of the noughts and crosses programs, i.e. the parts that select and make moves. We include the task structure and prototypical predicates for all the included parts of the two demonstration programs. The code for this demonstration program is shown in Appendix B.

The analysis process is traced in detail for both the program structures and the data structures.

6.1.2 Demonstration 2: A Program With an Introduced Error

The idealised program in the previous demonstration is altered so that it contains one sub-task whose implementation is based on an unusual and flawed design found in one of the student's programs. The recognition system does not contain any prototypical predicate that corresponds to this implementation. The program is found in Appendix C.

This demonstration illustrates:

- The use of hints to deal with unrecognised code;
- The clausal split mechanism.

6.1.3 Demonstration 3: The Original Study Set

This demonstration shows in detail the problems of identifying a single task, one of its sub-tasks and the related data representations within a study set of 16

programs. The analysis system is supplied with the task definitions, prototypical predicates and data structure information to recognise this task within all the programs in the study set.

The study set consists of the nine programs originally used in the study of design errors in Chapter 3, extended by a further seven programs taken randomly from the thirty MSc students' programs. These programs represent a wide range of solution strategies. From these sixteen programs, we have drawn our set of tasks, techniques and data structures that we have represented in the implemented system.

This demonstration illustrates:

- The variety of approaches and implementations that must be represented;
- The system's ability to identify the implementation of a particular task when the implementation is embedded in a program which does not correspond to any recognisable implementation of other tasks.

6.1.4 Demonstration 4: An Unfamiliar Set of Student Programs in the Same Domain

In order to test the success of our system with novel programs, we have kept a further set of seven programs which we did not inspect during the development of our system. These programs are also student programs in the domain of noughts and crosses game playing.

The analysis system is intended for use in situations where we do not expect to know all the ways of encoding a task. The analysis relies on the context to offer hints about unfamiliar implementations. It would be too severe test to expect the system to work successfully on unfamiliar versions of a task for which we have not encoded information about the context.

We include this test because the framework for representing programs is very flexible, so flexible that we could potentially build every task, programming

technique, and data structure for every program we encounter explicitly into the analysis system. This test gives an initial assessment of what further development is needed to recognise a realistic range of programs, and which improvements would be of most immediate use.

This test assesses

- how fully we have covered the possible implementations of that task, based on our analysis of the initial 16 programs;
- how well the program deals with novel implementations.

The objective is to identify a single task and the techniques used to implement that task, and the data structures and domain objects, as in the previous demonstration. The system is supplied with the same task definitions, prototypical predicates and information about data representations as were supplied for the previous demonstration.

6.2 Scope of Demonstrations

Scale The large number of tasks and the even larger number of different implementations makes it impossible in the time–frame of this project to supply enough prototypes and task declarations for all the variants of the whole exercise. Instead, for each of these demonstrations we trade off the number of tasks and domain objects we try to identify against the range of implementations, as shown in Figure 6–1.

The first and second demonstration show how the system operates to recognise many tasks and domain objects. To make these demonstrations feasible they are run on demonstration programs that were created for the purpose and therefore have little variation in implementation. The third and fourth demonstrations show systems behaviour over a full range of implementations. To make this feasible, these demonstrations cover only two tasks and three domain objects.

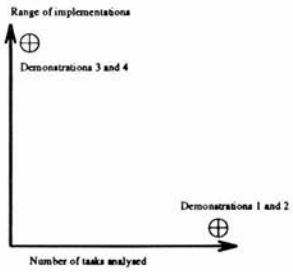

Figure 6–1: Scale of demonstrations

Inputs to the system All the demonstrations are run with the same set of definitions for tasks, prototypical predicates, data structures and domain objects. A single implementation is included for tasks and domain objects that are used only in Demonstrations 1 and 2. A range of implementations are included for the tasks and domain objects that are also used in Demonstrations 3 and 4.

Generality We have confined our demonstrations to a single programming exercise, the noughts and crosses program. There are some issues which, although interesting, we do not cover in these demonstrations.

These include an exploration of the generality of the architecture. In principle, this system can be applied to a new task domain (i.e. a new programming exercise) by replacing task declarations, and if the tasks require new programming techniques, by extending the set of prototypical predicates.

It is intended that prototypical predicates should be general across programming tasks, as are the programming techniques that they embody. The extent to which we have succeeded could be assessed by applying this system to students' implementations for a different programming exercise and seeing what changes to the task structure and extensions to the set of prototypical predicates are required.

To do this would require a hand analysis of the task structure of programs in a different programming exercise. Given the limitations of the system revealed in the demonstrations, such an assessment would be premature.

6.3 Demonstration 1: Overall Operation of the System

This demonstration illustrates the operation and features of the system by following its execution on a single program. The program is an incomplete noughts and crosses program, in which the top-level code, the sub-programs for choosing and making moves and changing players have been fully implemented, but other necessary sub-programs (i.e. initialising the board, displaying the board, testing for the end of the game and announcing the result) have not been implemented. The complete text of this progam is given in Appendix B and the task and prototype definitions used for the analysis are given in Appendix E Section E.1.

6.3.1 Loading and Preliminary Analysis

Loading

One student's program is analysed at a time. The call

```
ty_load(File).
```

reads the noughts and crosses code and stores the program, clause by clause, in a separate area of the Prolog database.

Call Tree

The system first computes and displays the complete call tree for the student's code. The system annotates the call tree according to whether each predicate

that is called is defined in the student's program or is undefined. Some parts of the demonstration program have been omitted and these are labelled in the output as UNDEFINED. Calls to Prolog system predicates are omitted.

```
** Call Tree for User Predicates
  1 play/1  % from user
  2    initialise/2  % UNDEFINED
  3    display_game/2  % UNDEFINED
  4    play/3  % from user
  5       end_state/2  % UNDEFINED
  6       announce/1  % UNDEFINED
  7       choose_move/3  % from user
  8          i_win/3  % from user
  9             fill_a_pair/3  % from user
 10                find_line/2  % from user
 11                pair_and_blank/3  % from user
 12                   empty_square/1  % from user
 13          block_lose/3  % from user
 14             next_player/2  % from user
 15             fill_a_pair/3  % see 9
 16          next_empty_square/2  % from user
 17             next_empty_square/3  % from user
 18                empty_square/1  % see 12
 19                next_empty_square/3  % see 17
 20       move/3  % from user
 21          move/4  % from user
 22             move/4  % see 21
 23       display_game/2  % UNDEFINED
 24       next_player/2  % see 14
 25       play/3  % see 4
```

Domain-Independent Type Inference

Data types are derived for each predicate that has been loaded. The data types that we use are specific to Prolog rather than to the applications domain.

During the type inference process, the data flow between variables in different predicates is used to infer a type for each predicate. As the type analysis is performed, the inference mechanism reports on all the predicates that contain data structures that cannot be typed. In this example, only the missing predicates present a problem to the type inference system. Our type inference process is

bottom up, and so we do not derive any type information for the arguments to the missing predicates. In other examples, contradictory inferences may be reported.

```
** No type info found for initialise/2
** No type info found for display_game/2
** No type info found for end_state/2
** No type info found for announce/1
```

Following the type inference, the types for all the predicates in the user's program are displayed. No information is displayed for the missing predicates. Wherever type information is needed from missing predicates or from predicates for whose arguments contradictory type inferences have been made, they are treated as if the heads and bodies of the undefined predicates create no data flow between their arguments and they leave the data types of their arguments unchanged.

```
** Inferred Types for User Code
empty_square(atom_type)
play(A)
find_line(list(B),
          triple_type(pair_type(B,integer),
                      pair_type(B,integer),
                      pair_type(B,integer)))
next_empty_square(list(atom_type),integer)
next_player(atom_type,atom_type)
block_lose(list(atom_type),atom_type,integer)
choose_move(list(atom_type),
            atom_type,
            pair_type(integer,atom_type))
fill_a_pair(list(atom_type),atom_type,integer)
i_win(list(atom_type),atom_type,integer)
move(pair_type(integer,C),list(C),list(C))

next_empty_square(list(atom_type),integer,integer)
pair_and_blank(triple_type(pair_type(atom_type,D),
                           pair_type(atom_type,D),
                           pair_type(atom_type,D)),atom_type,D)
play(list(atom_type),atom_type,E)
move(integer,F,list(F),list(F))
```

The type information for each predicate is stored along with the predicate in the database.

During this initial type inference, the implicit constraints between type variables in different predicates are used. After the initial type inference, the type variables in each predicate are treated as if they are independent of the type variables in any other predicate. This means that the subsequent analysis can reason about each task independently of any connections between that task and the other tasks with which it is connected. Inferences about the connections between tasks are made in a separate phase. Lacking empirical evidence, it is debatable at what stage of analysis these connections should be made, whether it is better to try to recognise tasks independently and then hope to combine them consistently, or to use the constraints between tasks to prune the analysis early. We expect to deal with erroneous programs, and so we do not want errors in one task to have a major effect on the analysis of connected tasks. Therefore we have chosen to analyse each task independently and then try to combine the results.

That ends the pre-processing stage.

6.3.2 Identify Individual Tasks, Their Implementation and Domain Objects

The system then starts searching for the implementations of different tasks in the code. Each task is synthesised according to different prototypes, and then each synthesised task is matched to the student's code.

The system reports when code for each task is identified with the user's code. The names of the user predicates, the task and prototypical predicate are reported and the synthesised code is displayed for tracing purposes.

When a task is identified, the system may also make some inferences about the domain objects that are manipulated by that task. If the arguments to that task represent domain objects then the system reports the data type inferences that have been made about each domain object.

Figures 6–2 and 6–3 show examples of the display that is produced when a task is identified.

Recognition Node Created:

Prototype:	ONE OFF_TEST
Task:	*empty_square/1*
User predicate:	**empty_square/1**
Domain object:	SQUARE of type *atom_type*
Links:	None

Other Information Used:

Top predicate in synthesised code :	**q/1**
Type of argument in synthesised code :	Unconstrained
Type of argument in user's code:	*atom_type*

As reported:

```
*** Matched Prototype one_off_test   Task empty_square/1
*** to user predicate(s) empty_square/1

Type: ===q(A) ===

q($empty_square):- true.

Domain object square in empty_square/1
is given type atom_type by user predicate empty_square/1
```

Figure 6–2: Recognising a Task: Test for an empty square

Recognition Node Created:

Prototype:	TESTED_ELEMENT_POSITION_ACC
Task:	*find_empty_square/2*
User predicates:	`next_empty_square/2`
	`next_empty_square/3`
Domain objects:	PLACE of type *integer*
	BOARD of type *list(atom_type)*
Links:	Task *empty_square/2* to user predicate `empty_square/1`

Other Information Used:

Top predicate in synthesised code :	`p/2`
Types of arguments in synthesised code :	*list(A)* and *integer*
Type of argument in user's code:	*list(atom_type)* and *integer*

As reported:

```
*** Matched Prototype tested_element_position_acc
***        Task  find_empty_square/2
*** to user predicate(s) next_empty_square/2, next_empty_square/3

Type: ===p(list(A),integer) ===

p(A,B) :-  q(A,1,B).

q([A|B],C,C) :-  empty_square(A).
q([A|B],C,D) :-  E is C+1,
                 q(B,E,D).

Domain object place in find_empty_square/2
is given type integer by user predicate next_empty_square/2

Domain object board in find_empty_square/2
is given type list(atom_type) by user predicate next_empty_square/2
```

Figure 6–3: Recognising a Task: Find the next empty square

Task	Prototype	User predicate(s)
play_game/1	GAME_TOP	play/1 play/3
move/3	REPLACE_NTH	move/3 move/4
choose_move/3	CHOOSE_MOVE_HEURISTIC	choose_move/3
i_win/3	METHOD WIN_IN_ONE	i_win/3
block_lose/3	BLOCK LOSE_NEXT_MOVE	block_lose/3
pair_and_blank/3	PAIR_AND_BLANK	pair_and_blank/3
line_next_move/3	FILL_A_PAIR_IN_SQ	fill_a_pair/3
find_line/2	FIND_LINE_IN_SQUARES	find_line/2
find_empty_square/2	TESTED_ELEMENT_POSITION_ACC	next_empty_square/2 next_empty_square/3
empty_square/1	ONE_OFF_TEST	empty_square/1
next_player/2	EXCHANGE_TWO	next_player/2

Figure 6–4: User Predicates Identified in Demonstration 1

When all the tasks that can be identified have been identified, the system reports on all the user predicates that have been identified with tasks (Figure 6–4).

The system also comments on any user predicates that have not been identified with tasks. In this example, all predicates have been identified.

6.3.3 Generic Variables

Some tasks include code that we do not wish to specify in full. Generic variables allow us to match any Prolog term regardless of its form. For example, the task of testing for an empty square includes a generic variable `$empty_square` which is matched to the atom `empty` in the user's code.

We report on the values of all generic variables that have been bound during the matching process. This is not strictly necessary since this information overlaps with the information that is derived for domain objects, and it is not integrated with the other information about domain objects. We report generic variables at this stage merely as a trace.

```
In task next_player/2 prototype exchange_two
     $tokenA is bound to o
     $tokenB is bound to x
```

```
In task empty_square/1 prototype one_off_test
     $empty_square is bound to empty
```

6.3.4 Combining Fragments of Code Structure Analysis

When the system has identified the code with as many tasks as possible and created recognition nodes for those tasks, it combines the analyses of tasks. This is done by following up the links between recognition nodes. If a predicate calls another predicate and the two predicates have been analysed as different tasks, then the task that corresponds to the calling predicate must call the task that corresponds to the called predicate. The links between recognition nodes indicate which matches are expected. If they do not, then our analysis contains an inconsistency and any such inconsistencies are reported (see Section 6.4.6 which describes an example of how we deal with inconsistencies). This example contains no such inconsistencies.

As the system checks the connections between tasks for consistency, it adds up the sizes of the combined fragments of code. Size is measured by scoring one for the task and adding that to the sum of scores for each of its linked sub–tasks. A task may be identified even though some of its sub–tasks are not. We also count the number of calls to unidentified sub–tasks in each task. Each unidentified subtask in a task is counted as a *hole*. The number of holes for a task is counted recursively through its sub–tasks in the same way as completed sub–tasks. Figure 6–5 shows the results.

6.3.5 Commenting on Hints for Unrecognised Code

At this stage, the system displays any hints it has obtained for code that has not been recognised. These hints may be used to drive further recognition. In this example, all the code that has been provided has been matched, so there are no hints.

Task	Complete	Size	Holes
play_game/1	No	17	4
move/3	Yes	1	0
choose_move/3	Yes	14	0
i_win/3	Yes	5	0
block_lose/3	Yes	6	0
pair_and_blank/3	Yes	2	0
line_next_move/3	Yes	4	0
find_line/2	Yes	1	0
find_empty_square/2	Yes	2	0
empty_square/1	Yes	1	0
next_player/2	Yes	1	0

Figure 6–5: Code Fragments Identified in Demonstration 1 With Their Sizes

6.3.6 Further Analysis of Unrecognised Code

If there is any unrecognised code, the system uses the hints to analyse it further. There is no unrecognised code in this example.

6.3.7 Combining Fragments of Data Structure Analysis

During the combination of small tasks into larger fragments, the analysis gathers information about the domain objects that are manipulated by the tasks in the student's program.

The system gathers together the type information for each domain object that has been inferred for each task that uses that domain object (Figure 6–6). Then it checks the type and parts of each domain object to infer the descriptor for that domain object (Figure 6–7).

The system reports on any conflicting inferences about the types or descriptors for domain objects. There are no conflicts in this example so none are reported.

6.3.8 Termination

The analysis stops at this point.

Domain object	Data type	Tasks
SQUARE	*atom_type*	*empty_square/1*
RESULT	Uninstantiated	*play_game/1*
LINE	*triple_type(* *pair_type(atom_type, integer),* *pair_type(atom_type, integer),* *pair_type(atom_type, integer))*	*find_line/2*
PLAYER	*atom_type*	*next_player/2* *choose_move/3* *line_next_move/3*
PLACE	*integer*	*find_empty_square/2* *line_next_move/3*
TO_MOVE	*pair_type(integer,atom_type)*	*choose_move/3* *move/3*
BOARD	*list(atom_type)*	*find_line/2* *find_empty_square/2* *choose_move/3* *line_next_move/3* *move/3*

Figure 6–6: Data Types for Domain Objects Identified by Tasks

Domain object	Data type	Descriptor
SQUARE	*atom_type*	*atom_square*
RESULT	Uninstantiated	No descriptor
LINE	*triple_type(* *pair_type(atom_type, integer),* *pair_type(atom_type, integer),* *pair_type(atom_type, integer))*	No descriptor
PLAYER	*atom_type*	*atom_token*
PLACE	*integer*	*numerical_id*
TO_MOVE	*pair_type(integer,atom_type)*	*place_and_player*
BOARD	*list(atom_type)*	*list_of_squares*

Figure 6–7: Data Types and Descriptors Identified for Domain Objects

6.3.9 Discussion

Recognising Tasks and Techniques

Tasks and prototypical predicates are identified with the predicates in the code that fulfil those tasks. Identifying programming techniques requires further development, since a single prototype may consist of several programming techniques. An extension to this system would classify the prototypes according to the techniques that they use.

Using Program Structure and Behaviour

The decision about which predicates correspond to which tasks is behavioural as well as structural. The type behaviour of predicates is used to limit search during analysis and also as a correctness check during synthesis.

Types are used during analysis, as an initial check on whether the code that is synthesised from a prototypical predicate will match a user's predicate. It is expensive to find a task, synthesise code for the task using some prototype, and compare the synthesised code to the student's code. Therefore type information is used as an initial check on the prototype before the remainder of the match is tried. The types for the arguments to the prototype are at least as general as the types for the predicates that will be synthesised from that prototype for any task. If the types of arguments to the student's predicate cannot unify with the types of arguments to the prototypical predicate, then the code that is synthesised from that prototype will not match the student's predicate. Hence an initial check is performed between the argument types of each prototype and the argument types of the student's predicate before synthesis and matching can proceed.

Types are also used during synthesis as a check on the correctness of synthesised code. Type inference in performed on the code as it is synthesised, and only code that is consistently typed can be synthesised. This is not a complete check that

the synthesised code correctly performs that task, but it is a partial check against synthesising code that is gibberish. The use of data types as specifications in the synthesis process could be explored further. We could declare that particular tasks impose particular restrictions on the type of their arguments, and require that only the synthesised code that follows those declarations be considered a valid implementation of the task. In the present system, the synthesised code is required only to be typed consistently and to have the same number of arguments as the task.

Combining Fragments of Analysis

The intention behind recording the size of connected fragments is to use them to choose between competing interpretations of the code. At present the size of fragments is recorded but is not used.

Our analysis measures the size of connected fragments in terms of the number of tasks within each connected fragment. There is an argument for extending the size measurement so that fragments are measured not only in terms of the number of tasks recognised, but also in terms of the size of each task. For example we could measure the number of lines of code or predicate calls that the fragment accounts for. Then the system could at an early stage prefer a match to a "larger" task over a match to a smaller one. Such a measurement would allow us to prefer analyses that give larger consistent connected fragments over analyses that only give small ones.

6.4 Demonstration 2: Operation of a Clausal Split Transformation

In Section 5.3 we have described the use of clausal split to localise a piece of unrecognised code. Here we present a program in which the predicate which finds an empty square has been written incorrectly. The code is otherwise identical to

the code used for Demonstration 1. The incorrect part of the code is presented for reference in Figure 6–8. The process has been outlined in Section 5.3.1.

Student P12's Code to Find the Position of an Empty Square

```
next_empty_square(L, C) :-
     next_empty_square_sub(L, [1,2,3,4,5,6,7,8,9], C).
next_empty_square([H|_], [C|_], C) :-
     is_empty(H).
next_empty_square([_|T], [_|Cs], C) :-
     next_empty_square(T, Cs, C).
```

Figure 6–8: Example of Bodged Prolog Code

The first steps of the analysis proceed in the same way as for Demonstration 1.

6.4.1 Loading and Preliminary Analysis

The call tree and domain-independent type inference for all predicates are identical to those in Demonstration 1.

6.4.2 Identify Individual Tasks

The tasks and their domain objects are identified in the same way as for Demonstration 1 (e.g. Figure 6–2). Only the task *find_empty_square/2* (Figure 6–3 and Figure 6–4) is not identified. The user predicate **next_empty_square/2**, which is identified with task *find_empty_square/2* in Demonstration 1 is not identified with any task.

6.4.3 Generic Variables

Generic variables are identical to Demonstration 1.

Task	Complete	Size	Holes
play_game/1	No	15	5
move/3	Yes	1	
choose_move/3	No	12	1
i_win/3	Yes	5	
block_lose/3	Yes	6	
pair_and_blank/3	Yes	2	
line_next_move/3	Yes	4	
find_line/2	Yes	1	
empty_square/1	Yes	1	
next_player/2	Yes	1	

Figure 6–9: Code Fragments Identified in Demonstration 2 With Their Sizes

6.4.4 Combining Fragments of Code Structure Analysis

The system finds that the analysis so far is indeed consistent. The scores for the size of the code are rather different, since the *find_empty_square/2* task has not been identified (Figure 6–9). The *choose_move/3* task which uses *find_empty_square/2* therefore has one "hole", and *play_game/1* which uses the incomplete *choose_move/3* task has an additional hole.

Both tasks are smaller by two sub-tasks, not just one. The size mechanism measures the largest connected fragment of the task structure. The task *find_empty_square/2* has a sub-task *empty_square/1*. The sub-task was recognised, but the connecting task was not recognised and so it did not contribute to the size of the parent tasks.

6.4.5 Commenting on Hints for Unrecognised Code

At this stage, the system displays any hints it has obtained for code that has not been recognised. These hints may be used to drive further recognition. In this example, the recognition of *choose_move/3* implies that there should be a *find_empty/2* sub-task but the code has not been matched.

```
Task choose_move/3 calls sub--task find_empty_square/2.
```

Tested Element with Tail Recursive Counter

TESTED_ELEMENT_POSITION_ACC

Join Description
Join Expressions:
$position(List, Count) \Leftarrow test_1(List) \bowtie count_1(Count)$
$position(List, Acc, Count) \Leftarrow test_2(List) \bowtie count_2(Acc, Count)$
Component Prototypes:
$test_1, test_2$ EXPAND(STEPPER)
$count_1, count_2$ ENUMERATING_COUNTER

Prototypical Predicates (may be split):

```
      p(List, Pos)  :-  B(Base), q(List, Base, Pos)
q([H|T], Pos, Pos)  :-  T (H)
  q([H|T], P, Pos)  :-  I (P, Next), q(T, Next, Pos)
```

Figure 6–10: A Prototype That May Be Split

```
This call matches a call to the user's predicate next_empty_square/2
but the user's code for next_empty_square/2 does not match
any realisation of this sub--task.
```

6.4.6 Further Analysis of Unrecognised Code: Clausal Split

The system has some unrecognised code and a hint about how to analyse it. The hint suggests that the user's predicate **next_empty_square/2** should match code generated from the task *find_empty_square/2*. The system identifies which of the prototypes that can be used to realise this task can be split into components, and how they can be split.

Prototypes that can be split are identified by a join description that specifies a join expression and names the component prototypes.

Select a prototype for decomposition The system finds a prototype TESTED_ELEMENT_POSITION_ACC, for which there exists a join description (Figure 6–10).

The system first confirms that the prototype TESTED_ELEMENT_POSITION_ACC could be used to implement the task *find_empty_square*. The system has already synthesised the code for all the prototypes that can be applied to this task, and so the applicability of the prototype for this task is confirmed simply by looking up the realisation for that task and prototype.

Synthesise code for components Having confirmed that the TESTED_ELEMENT_POSITION_ACC is a candidate, the system then synthesises code for the task using each of the component prototypes.

The first prototype is a variation of an existing prototype (STEPPER), but the variation is not stored explicitly. The variation is created according to the join specification. The operation EXPAND takes the prototype it is given as an argument and nests it in a predicate call.
Given the prototypical predicate for STEPPER:

```
stepper([H|_]) :- T(H).
stepper([_|T]) :- stepper(T).
```

The prototypical predicates for EXPAND(STEPPER) are:

```
stepper1(A) :- stepper(A).

stepper([H|_]) :- T(H).
stepper([_|T]) :- stepper(T).
```

The following code is synthesised for *find_empty_square/2* from EXPAND(STEPPER):

```
stepper1(A):- stepper(A).

stepper([A|B]):- empty_square (A).
stepper([A|B]):- stepper(B).
```

The second component is an enumerating counter. Given the the prototypical predicate for ENUMERATING_COUNTER:

```
counter1(A):- B(B), counter(B, A).

counter(A, A):- true.
counter(A, B):- I(A, C), counter(C, B).
```

The following code is synthesised for *find.empty_square/2* from ENUMERATING.COUNTER:

```
counter1(A):- counter(1, A).

counter(A, A):- true.
counter(A, B):- C is A + 1, counter(C, B).
```

All the sub-tasks in the task *find.empty_square/2* can be assigned either to one component prototype or to the other, so there is no need to consider splitting each sub-task.

Clausal split of the user's predicates Next the user's predicates `next_empty_square/2` and `next_empty_square/3` are split according to the two join expressions.

The first join expression gives:

```
s3(A):- s1(A).
s4(A):- s2([1,2,3,4,5,6,7,8,9],A).
```

while the second gives

```
s1([A|B]):- s1(B).
s1([A|B]):- is_empty(A).

s2([A|B],C):- s2(B,C).
s2([A|B],A):- true.
```

Distributing the results of the two splits into the right predicates, the two components of the user code are:

```
s3(A):- s1(A).

s1([A|B]):- s1(B).
s1([A|B]):- empty_square(A).
```

and

```
s4(A):- s2([1,2,3,4,5,6,7,8,9],A).

s2([A|B],C):- s2(B,C).
s2([A|B],A):- true.
```

Identify the components The system tries to identify the synthesised components with the results of the split on the user's code. In this example, only one of the two components can be identified: the expanded list stepper that tests for an empty square. The other component, the enumerating counter, is not.

The criterion used for whether or not the components of a user predicate match the components of a task is that if one or more components match, then the match is considered a success. By this criterion the system infers that the code is indeed intended to implement the `find_empty_square/2` task, but that it is bodged. The system reports that it has identified the task, and which part of the implementation was bodged. It also presents the prototypical code for the task as a correct implementation, as follows:

```
Procedure next_empty_square/2 appears to fulfil the task
find_empty_square/2 but is bodged.

The code approximates to method tested_element_position_acc but the
enumerating_counter is unfamiliar.

A correct implementation of procedure next_empty_square/2 would be:

    p(A,B) :- q(A,1,B).

    q([A|B],C,C):- empty_square(A).
    q([A|B],C,D):- E is C+1, q(B,E,D).
```

Even though the predicates are bodged, the types are for domain objects in this task are inferred from the user's code.

There are other possible realisations for *find empty_square*. If the first split had not matched, and if any other prototypes for these realisations had been able to be split, then they would also have been tried.

If there is more than one such hint the system tries to satisfy each hint in turn.

6.4.7 Continuation

Using the new analysis, the system repeats the procedures for **combining fragments of code structure analysis**, **commenting on hints for unrecognised code**, and **further analysis of unrecognised code**. The consistency of the expanded analysis is checked and any further hints that arise from it are dealt with. In this example, there are no more hints and no further analysis.

The remainder of the analysis (**Combining fragments of domain structure analysis**) proceeds exactly as for Demonstration 1.

6.4.8 Discussion

At present, the description of what has gone wrong is little more than a trace of the system's own workings and could be improved. The internal names of prototypes, which are used as part of the description of what has gone wrong, are not in themselves very useful to the student! As a first step, the prototypes could be rewritten to use more helpful predicate names.

The system could produce more a more useful commentary if it maintained more information about the programming techniques that the prototypes represent and the roles within those prototypes. This information could include the fact that this enumerating counter is one instance of a class of counters and that it uses an accumulator technique. Suitable information about roles might be that that the counter requires a base (corresponding to $\mathcal{B}$) and an increment (corresponding

This table illustrates the range of different task structures and implementations for a single task, that of finding an empty square.
The first column reflects the general approach to the problem. The first approach is to identify an empty square, without actually moving to it. The second approach actually tries to "take" the square. The third approach maintains or creates a list of empty squares and picks the head from this list.
The second column distinguishes between recursive and non-recursive implementations. The third column distinguishes between different representations for the board. The implementations that are used vary depending on the shape of the board.

Approach	Technique	Board	Program
Find empty square	Recursive	List of squares	P1 P5 P8 P10 P12 P13
		List of lines	P3 P6
	Non-recursive	Term of squares	P15
Move and test	Recursive	List of squares	P14 P16
	Non-recursive		P11
Pick from empties list	List head	List of rows	P4 P7
		Nested lines	P9
Unidentifiable			P2

Figure 6–11: Find empty square: Summary

to $\mathcal{I}$). Explicit links between the prototype and the techniques that it embodies might also form the basis of a student model, noting that the student had a problem with a prototype that uses accumulating and counting techniques.

6.5 Demonstration 3: The Study Set

This demonstration shows how the system recognises the various implementations of a single task within a previously studied set of programs. The task of finding an empty square was chosen because it is a comparatively simple task, it is specific to the noughts and crosses problem, and it most of the students found it necessary to create a separate task to do this at some point in their programs.

We have tested the system on 16 student's implementations of the task of finding any empty square. We inspected all the programs and supplied a sufficient set of prototypical predicates and task definitions to recognise them. Figure 6–11 describes the range of implementations. We found very different general

approaches to the problem, including searching the board structure for an empty square, trying to take each square in turn until the program successfully takes an empty square, and maintaining or creating a list of the positions of all the empty squares. Within these general approaches, the implementations vary according to which data structures have been used to represent the board and according to whether the students have chosen a recursive or a non–recursive technique. Even when implementations have been classified as similar according to these criteria they still vary considerably. None of the implementations have an identical form.

The system does not have program transformation rules which can reason about the equivalence of programs. A variety of program transformation approaches have been dealt with in other systems (Looi, 1988b; Murray, 1988; Gegg-Harrison, 1992), and we do not deal with them here. As a result, we need to perform some transformations by hand.

There are two groups of transformations. The first group preserve the meaning (i.e. the input–output behaviour) of the code. These transformations are purely mechanical, and could easily have been implemented at a cost of increased search. These transformations are to re–arrange arguments, to divide 'or' branches into separate predicates, and to fold or unfold predicates to make the predicate structure correspond more closely to the structure of the synthesised code. These transformations could be made using rules that are independent of the purpose of the code and the context in which it is called.

The second group of transformations change the meaning of the code. They are to remove extraneous control (typically cuts), to correct buggy code, and to ensure that programs have the right number of arguments to correspond to the specification. These transformations require prior knowledge or hypotheses about the intended behaviour of the student's code in the context within which it is called, so that the changes to the behaviour of the code are appropriate. To make these changes automatically would require rules that take into account the purpose of the code and the context in which it is called.

This is the complete list of transformations that have been performed by hand on these programs before the automated analysis:

Re-arrange arguments to match the order of synthesised arguments in predicates and data structures.

Divide 'or' branches into separate predicates.

Fold and unfold predicates to make the predicate structure explicit.

Change lists into terms if lists were used as records.

Remove extraneous control (typically cuts).

Rewrite bugs Buggy code has been corrected.

Correct numbers of arguments that don't fit specification Ensure that the arguments to predicates in the specification matched those in the specification.

The code for the 16 students' versions of finding an empty square is shown in Appendix D. Where the code has been transformed by hand, the original and transformed versions are both shown. The prototype definitions and task definitions are shown in Appendix E Section E.2.

6.5.1 Code Structure Analysis

The tables in Figures 6–12, 6–13, 6–14 and 6–15 show the results of the automated analysis of tasks and prototypical predicates. The analyses are grouped according to the summary in Figure 6–11. The tables include the task of finding an empty square as well as any sub–tasks involved. The task and prototype definitions that were used in the previous demonstrations are also included. Where these match the students' code, they are also presented in these tables.

As expected, the automated analysis of finding an empty square is effective in all the programs apart from P2 (in which even our own inspection did not identify the task of finding an empty square).

We also included the task and prototype definitions for other tasks in the noughts and crosses program as they were used in the previous demonstration. We did

Tasks and prototypical predicates in student programs to find an empty square.

	Task	Prototype	Predicates
P1	*find_empty_square/2*	TESTED_ELEMENT_POSITION_GT	`poss_move/2` `pick_nth/3`
P5	*empty_square/1*	ONE OFF_TEST	`empty_square/1`

	find_empty_square/2	TESTED_ELEMENT_POSITION_ACC	`next_mv/2`
			`first_empty/3`
			(bodged)

P8	*empty_square/1*	ONE OFF_TEST	`empty_square/1`
	find_empty_square/2	LOOKUP.MEMBER	`find_empty_square/2` `member/2`
P10	*empty_square/1*	ONE OFF_TEST	`empty_square/1`
	find_empty_square/2	MATCHED_ELEMENT_POSITION	`empty_position/2` `position_in/3`
P12	*empty_square/1*	ONE OFF_TEST	`empty_square/1`

	find_empty_square/2	TESTED_ELEMENT_POSITION_ACC	`free_square/2`
			`free_square/3`
			(bodged)

P13	*empty_square/1*	ONE OFF_TEST	`empty_square/1`
	find_empty_square/2	TESTED_ELEMENT_POSITION_ACC	`any_move/2` `any_move/3`

Figure 6–12: Find Empty Square: Recursive Search in a List of Squares

not try to cover the full range of implementations for these tasks. The top–level task *play_game* was identified in 9 of the 16 programs. The specification did not state that the top–level predicates were to be left unchanged and so the other 7 students altered them so that they could not be matched. The task for changing players *next_player/2* was identified in 8 of the programs. A small number of definitions for general–purpose tasks were also included, such as appending two lists and reversing lists. Three implementations of *append/3* were identified.

Tasks and prototypical predicates in student programs to find an empty square. (contd)

Program	Task	Prototype	Predicates
Recursive Search in a List of Squares:			
P3	*empty_square/1*	VAR_TEST	`empty_square/1`
	find_empty_square/2	NESTED_TEST	`any_space/2` `any_move/2`
P6	*empty_square/1*	INTEGER_TEST	`empty_square/1`
	find_empty_square/2	NESTED_REC_TEST	`avail_winset/2` `avail/2`
Non-Recursive Search in a Term of Squares:			
P15	*empty_square/2*	INDEX_AND_TEST	`legal_move/2`
	find_empty_square/2	MATCH_AND_TEST	`pick_move/2`
	pick_any_square/2	PICK_NINE	`best_move/2`

Figure 6–13: Find Empty Square: Other Search Approaches

Tasks and prototypical predicates in student programs to find an empty square. (contd)

Program	Task	Prototype	Predicates
Board is a List of Rows:			
P4	*empty_square/1*	ONE_OFF_TEST	`center/1`
	find. empty_square/2	GET_ONE_FROM_LIST_IN_RECORD	`first_empty/2`
	access_empties/2	ACCESS_FOUR_OF_FIVE	`empties/2`
P7	*empty_square/1*	ONE_OFF_TEST	`empty/1`

	find. empty_square/2	GATHER_POSITIONS	`empty_square/2`
			`getx/3`
			`belong/4`

Board is a Nested Term of Lines:			
P9	*empty_square/2*	INDEX_AND_TEST	`empty_sq/2`
	find. empty_square/2	COLLECT_AND_SORT	`any_move/2`

Figure 6–14: Find Empty Square: Take the Head of a List of Empty Squares

Tasks and prototypical predicates in student programs to find an empty square. (contd)

Program	Task	Prototype	Predicates
Recursive Approach			
P14	*empty_square/1*	INTEGER_TEST	`empty_square/1`
	choose_move/3	MOVE_AND_ASSESS	`find_move/3`
	move/3	REPLACE_TEST_OR_MATCH	`move/3`
P16	*empty_square/1*	INTEGER_TEST	`empty_square/1`
	choose_move/3	MOVE_AND_ASSESS	`good_move/3`
	move/3	REPLACE_TEST_OR_MATCH	`move/3`
Non-Recursive Approach			
P11	*empty_square/1*	ONE_OFF_TEST	`empty_square/1`
	choose_move/3	MOVE_AND_ASSESS	`select_move/3`
	move/3	REPLACE_NONREC_NINE	`move/3`

Figure 6–15: Find Empty Square: Move and Assess Approach

6.5.2 Data structure analysis

The following tables show the results of the data analysis for the domain objects involved in the task of finding an empty square.

Figure 6–16 describes the board. Figure 6–17 describes how squares are represented, as derived from the test for an empty square.

This table illustrates the range of different data structures and representations for the board domain object as it occurs in the task of finding an empty square.

Domain Object	BOARD	
Program	Data Type	Representation
P1	*list(_)*	Not described
P2	Task not recognised	
P3	*list(triple_type(T,T,T))*	Not described
P4	*quin_type(_,_,_,list(_),_)*	Not described
P5	*list(atom_type)*	*list_of_squares*
P6	*list(list(integer))*	Not recognised
P7	*list(list(atom_type))*	Not recognised
P8	*list(atom_type)*	*list_of_squares*
P9	*triple_type(_,_,_)*	Not described
P10	*list(atom_type)*	*list_of_squares*
P11	*list(atom_type)*	*list_of_squares*
P12	*list(atom_type)*	*list_of_squares*
P13	*list(atom_type)*	*list_of_squares*
P14	*list(integer)*	*list_of_squares*
P15	_ (unidentified)	
P16	*list(integer)*	*list_of_squares*

The board types of P11, P14 and P16 have been obtained from the *choose_move/3* task. The other board types have been obtained from the *find_empty_square/2* task.
The BOARDs in P5, P8, P10, P11, P12, P13, P14, P16 are recognised as *list_of_squares*. The types *atom_type* or *integer* have been identified as representing squares in the task *empty_square/1*.
The BOARDs in P6, P7 are not recognised as *list_of_lines* because no task has been identified in their code which explicitly deals with lines. Analysis of the other tasks in thee programs would supply this information.
The type of the BOARDS in P15 was not sufficiently instantiated to fulfill the type constraint.
The types of the BOARDSs in P1, P3, P4, P5, P9 were not sufficiently instantiated to fulfill the part constraint. Analysis of other tasks might instantiate this type information further.
The BOARD in P2 is not recognised because the task is not recognised. Analysis of other tasks might yield a description for the BOARD.

Figure 6–16: Find Empty Square: Summary of Board Information

This table illustrates the range of different data structures and representations for the SQUARE domain object as it occurs in the task *empty_square/1* which tests if a square is empty.
No information about SQUARE is obtained when the *empty_square/1* task has not identified.

Domain Object	SQUARE	
Program	Data Type	Representation
P1	Variable	Not described
P2	Task not recognised	
P3	Variable	Not described
P4	*integer*	*integer or filled_square*
P5	*atom_type*	*atom_square*
P6	*integer*	*integer or filled_square*
P7	*atom_type*	*atom_square*
P8	*atom_type*	*atom_square*
P9	Variable	Not described
P10	*atom_type*	*atom_square*
P11	*atom_type*	*atom_square*
P12	*atom_type*	*atom_square*
P13	*atom_type*	*atom_square*
P14	*integer*	*integer or filled_square*
P15	*atom_type*	*atom_square*
P16	*integer*	*integer or filled_square*

Figure 6–17: Representation of Squares

6.6 Demonstration 4: A Set of Unfamiliar Programs

For this demonstration, the analysis was run on a further set of seven students' programs. These programs were written to the same specification by students on the same course, but were from the following year. These programs were not inspected by the experimenter nor altered in any way before the automated analysis. The aim was to recognise the task of finding an empty square, using the same task structure and the prototypes that were derived for Demonstration 3.

The task of finding an empty square was not identified in any of the programs. We looked at the programs to find the reasons for this. Our results (shown below) indicate that in the present state of the system, the lack of transformations are significant but they are not the most significant factor in failing to recognise the task. For this reason, we did not go on to perform hand transformations on this sample.

New Strategy Three of the programs used a new strategy that we had not identified before. To recognise this implementation of the task would require the inclusion of a new prototype. This suggests that before the system could be used, a wider range of programs must be studied and prototypes for those implementations included. All three programs used the same strategy, which offers some hope that reasonable coverage could be achieved without an impossibly large number of new prototypes.

Lack of Transformations Two programs were variants of existing prototypes, and with appropriate transformations they would have been recognised.

Task Missing One program did not perform the task of finding an empty square at all. Instead, the program contained a set of patterns to represent all possible permutations of the board.

Non-working program One program did not run. This program expressed a set of board patterns but the code was not complete enough to be certain what was intended.

We conclude that the range of prototypes must be extended if the system is to be effective. Further study of noughts and crosses programs is needed to derive a representative set of approaches that students use for even this simple task. In addition to this, the difficulty in matching the prototypes we have shows that even given a large collection of prototypes, transformations to deal robustly with minor variants of implementations will still be needed for effective recognition.

6.6.1 Code Structure Analysis

Figure 6-18 shows that we have had little success in recognising the task of finding an empty square in these programs. Subsequent inspection of the code shows that the students have used different programming techniques for this task to those used in the code that had been inspected.

6.6.2 Data Structure Analysis

We are able to infer a considerable amount of data type information. Having failed to recognise the task of finding an empty square, we are unable to link the type information to the domain objects that are used in finding an empty square, and therefore we cannot derive information about data representations.

6.6.3 Discussion

Coverage

The difficulties in recognition were mainly due to the use of different programming techniques in different contexts to those found in the test set. As a result,

This table describes how the inference mechanism recognises tasks and prototypical predicates in student programs for finding an empty square.

Program	Task	Prototype	Predicates
P21	*next_player/2*	EXCHANGE_2	`next_player/2`
P22	*play_game/1*	GAME_TOP	`play/1` `play/3`
P23	*next_player/2*	EXCHANGE_2	`next_player/2`
	play_game/1	GAME_TOP	`play/1` `play/3`
P24	*next_player/2*	EXCHANGE_2	`next_player/2`
	play_game/1	GAME_TOP	`play/1` `play/3`
P25	*play_game/1*	GAME_TOP	`play/1` `play/3`
P26	*next_player/2*	EXCHANGE_2	`next_player/2`
	play_game/1	GAME_TOP	`play/1` `play/3`
P27	*append/3*	SCHEMA_A	`append/3`
	next_player/2	EXCHANGE_2	`next_player/2`
	play_game/1	GAME_TOP	`play/1` `play/3`
	empty_square/1 (misidentified)	ONE_OFF_TEST	`allplaces/1`

Figure 6–18: Recognition for Unfamiliar Programs

the library of prototypical predicates was not big enough to recognise the task as given.

Novel Implementations

Our recognition mechanism is very unforgiving of variations in the the code. Given the following code as synthesised for swapping players:

```
next_player(o, x).
next_player(x, o).
```

we cannot recognise that the following code is (more or less) equivalent to it:

```
next_player(Player,Player1):-
        (Player=o,Player1=x);
        Player1=o.
```

To recognise these two pieces of code as equivalent, we would require context-independent operators that could move unifications from the head of the predicate into the body and that could replace two clauses with a single clause using the semi-colon **or** construct. We would also require a context-specific operator that could reason about whether or not it is appropriate to treat the second **or** branch as a default case that ignores the value of the first argument.

The difficulty of using program transformation lies in determining when different programs should be described as different implementations of different algorithms and when they should be seen as trivial or surface variants of the same algorithm. Given enough transformation rules, it is possible to transform an implementation of one algorithm into an implementation of another (Gegg-Harrison, 1992). Looi used heuristics to decide which algorithm was being attempted. There remains an open philosophical question about what constitutes a significantly different algorithm versus what is merely a trivial variation in implementation. We have preferred to avoid this issue by insisting on very exact matching, although we are aware that a practical implementation would need to deal with it using for instance the techniques suggested by (Looi, 1988b; Murray, 1988; Gegg-Harrison, 1992).

Type Errors

In general, the type inference was successful, although we encountered three problems.

The type inference mechanism cannot deal with programs in which a single argument refers to more than one data type. This is typical of programs such as `flatten/2`, in which a single argument may be a type, or a list of elements of that type, or a list of lists of elements of that type, and so on indefinitely. One student wrote such a mechanism for manipulating a complex board structure.

The type inference mechanism cannot deal with programs in which operators are overloaded or are used in unexpected ways that conflict with the type declaration. For instance, operators such as / and - are expected to take integer arguments, but some students used them to construct records with other types of argument.

There was a common conflict between atomic and integer elements being put into the same list.

These three problems may all be seen as the result of using a type language based on particular conventions to describe a programming language in which data type conventions are not required.

Consequences for Domain Objects

The recognition of the domain objects is dependent on both the type inference mechanism and the task recognition being successful. The tasks are not recognised and so the domain objects are not recognised, either, even though we have successfully obtained some data types that are strongly suggestive.

Our results suggest that the analysis of data structures is more robust across the students' various implementations, and the analysis system would benefit from letting the analysis of data structures drive the analysis of program structure more than it does it present.

More broadly, our results suggest that for intention-based debugging, reasoning directly about program behaviour may provide a more robust source of information about the program and at a lower cost in search than other approaches that reason only from structure to behaviour.

Results

This demonstrates the need for:

- More prototypical predicates.
- The use of transformations to capture more variations of code.
- Making the recognition of data structures less dependent on the recognition of tasks.

6.7 Discussion

6.7.1 Performance

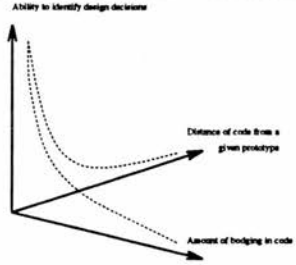

Figure 6–19: Effect of variations and bodges on recognition

To identify even a small task is in itself complex. Many small tasks are optional and do not appear in all of the programs, depending on the strategy that the

student has used to design the program. We find that the performance of the recognition system degrades rapidly as code deviates more from the prototypes (Figure 6–19). A large number of prototypes are needed to represent the many implementations of each task. We found that the prototypes for the task of recognising the empty square that we had obtained from our analysis (used in Demonstration 3) were not sufficient to recognise the same task in other programs (Demonstration 4), for which still more prototypes are needed. The variation is due both to implementations that we had not foreseen and to bodges that we had not foreseen.

6.7.2 Variations in the Predicate Structure

This analysis follows an intuition that a single task in the problem domain should, and typically would, be represented by a single call to a separate predicate. It is expected that distinct sub-tasks of the main task are represented by calls to other predicates. This intuition is not true for all the tasks in the students' programs. It was quite common that students would unfold sub-tasks into the body of the main predicate. Without a mechanism for automatically unfolding the synthesised code for testing an empty square into the body of the recognised code, the system recognises the following code:

```
empty_position(Board, Move) :-
    empty_square(Empty),
    position_in(Board, Empty, Move).

empty_square('*').
```

but it does not recognise the following variation:

```
empty_position(Board, Move) :-
    position_in(Board, '*', Move).
```

The `empty_square/1` predicate that represents the separate task of testing or creating an empty square has been folded into the body of the predicate that finds an empty square.

An unfolding mechanism could be provided, but it is not obvious when code should be unfolded. Some means of standardising both the user's code and the synthesised code by appropriate unfolding could be used to make the matching more general. A recording mechanism or else a further pass of the original program would be needed to diagnose whether the student had in fact unfolded the sub-task.

Our view of programs in which a single task is implemented by a single predicate has a rather different emphasis from the view that is given for procedural programming languages like Pascal (Spohrer *et al*, 1985). In Pascal, sub-tasks may be nested within a single procedure, represented by e.g. nested loops. It is also a rather different model to that of Proust, which gives great importance to the way that parts of different tasks may be interleaved in the structure of the code.

If a sub-task can only reasonably be implemented in one way, then the Prolog realisation for that task is simply included in the body of the code for the main task. If the code for a sub-task can be implemented in different ways, then the architecture requires that sub-task to be realised as a separate Prolog predicate. If the alternative implementations of the sub-task are very simple then there is no overriding reason why the sub-task should be written as a separate predicate, and it is quite reasonable for a student's implementation to incorporate the sub-task in the predicate for the main task. In effect, the architecture has conflated an *or* branch in the goal/plan graph with a choice of task.

6.7.3 Domain Specific and Domain Independent Tasks

The architecture has made a division between domain specific tasks and domain independent prototypes. Yet some tasks can be used in many problem domains. Tasks can be specific to the problem domain (such as finding an empty square, a task that is specific to board games) or they can be applicable to many problem domains (such as appending two lists, a task that is performed within many Prolog programs (Figure 6-20). These represent two different views of the same

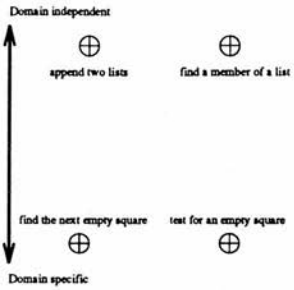

Figure 6–20: Domain Specific and Domain Independent Tasks

code, since code for a general task may implement all or part of a domain–specific task.

The emphasis in this work is on the detection of domain specific tasks. The architecture permits only one view of the tasks in a program, and our objective is to seek a domain–specific interpretation, so the mechanism has been supplied mainly with domain–specific tasks. When a general–purpose task is used as part of a domain–specific task, we have represented the general task in the prototype that is used for the more specific task. Wherever a piece of code could be given both a domain–specific interpretation and a domain–independent interpretation, the domain–independent task definition has been removed so that code is synthesised from, and matches, only the domain–specific version of the task.

If the same domain–independent code is used to implement several different tasks in the problem domain, then that code is only recognised as performing one such task. A useful extension to the system would represent both domain–specific and domain–independent tasks and provide a mapping between them.

6.7.4 Coverage

Our final demonstration shows that we have achieved only incomplete coverage of the prototypical predicates necessary for locating an empty square.

The coverage of data structures for this task is more complete. The main problem with data structures is dealing with those that the type inference mechanism cannot handle, such as lists whose elements are of different types.

For the successful recognition of a task in isolation, the prototypical predicates must cover all the variations in the way that programming techniques are used and combined. More detailed analysis of variations in the code, such as clausal split are attempted only when there is more information from the surrounding context. This information could be supplied by providing more detailed task structures and prototypical predicates for the other tasks in which finding an empty square is embedded.

6.7.5 Possible Extensions to the System

These demonstrations of the program indicate that the system could usefully be extended in a number of ways.

Tasks and Prototypes A larger library of tasks is needed to represent the parts of the noughts and crosses program that we have not investigated in detail. Even for the single task of finding an empty square, the range of prototypes must be extended if the system is to be effective. Further study of noughts and crosses programs is needed to derive a representative set of approaches that students use even for this comparatively simple task.

Representing programming techniques The system could produce a more useful commentary if it maintained more explicit information about the programming techniques that the prototypes represent. At present, the names of programming techniques offer a clue to the programming techniques that they embody, and the join specifications link composed prototypes to smaller prototypes that represent individual programming techniques. A formal classification of programming techniques (e.g. (Gegg-Harrison, 1991)) could be linked to the prototypes and used for tutoring.

Code variations Successful analysis depends on a close match between the task structure and programming techniques that are represented in the analysis system, and the predicate structure that is used by the student.

The coverage of the system would be improved by the addition of mechanisms for dealing with variations in the form of code. This could be done by transforming the synthesised code to equivalent forms in a principled way or else by using heuristics to look for a close match to a schema. It might be worthwhile to standardise the form of the code at least for initial recognition, and some of the hand transformations standardise the form of the code. The difficulty on standardising code before analysis is that the standardisation process itself may lose information about the students' intentions. This problem is perhaps worse in Prolog than in many other programming languages, where quite small changes to the surface structure of the program can lead to very different interpretations of the program technique being used (Looi, 1988b). A small variation in the order in which terms are called means that a generate and test technique is being used (inappropriately, in this example):

```
empty_position(Board, Move) :-
    position_in(Board, Empty, Move),
    empty_square(Empty).

empty_square('*').
```

The students' code does not necessarily map directly onto the task structure in the analysis system. The task structure leads to code being synthesised with a specific predicate structure, which may not correspond to the student's predicate structure. Synthesised predicates may in some sense exist in the students' code, but the student may have folded them into the body of the calling predicate.

Any transformation of the program loses some information about the student's original intentions in creating the program. This is most obvious when a transformation affects the behaviour of the program — how do we know that the new behaviour of the program is what the student intended? But even when the

behaviour is not affected, the structure of the program reflects the student's intentions about the task. We have started off from an initial working assumption that separate tasks are typically mapped onto separate predicates. Changing the call structure of the program would change our interpretation of which tasks the student is trying to perform and how the student perceives those tasks.

There would be major questions about how to control the search in the application of such transformations, and how to derive appropriate strategies for when such transformations should be applied.

Greater priority for data structure information The system is capable of recognising novel and erroneous implementations if the the surrounding context provides some clues for driving the recognition. When sufficient context was available, clausal split was useful. Where there is no additional context, when the system only has information to recognise a single task then the our system performs poorly on novel implementations of that task.

It would help greatly if the interpretation of data structures did not depend so much on the successful recognition of the predicates that use them. Data structure information is more easily obtained and could be used to drive the recognition of some predicates.

6.8 Summary

In this chapter, we have presented demonstrations of the system running and an evaluation of its performance. We demonstrated the way that the system recognises and combines program fragments. We demonstrated the operation of clausal join. We also evaluated the system on students' programs. We showed that it works successfully on the programs that have been studied, but that (not surprisingly) at works poorly on unfamiliar programs where context for analysis is lacking. We assessed its performance on other programs and used this

assessment to outline and prioritise the additional work that is needed to lead to a practical recognition system.

Chapter 7

Conclusions

7.1 Contribution

In this section, we describe our main contributions.

The context of the problem is that of trying to identify design decisions in terms of the programming language and in terms of the problem domain. We are extending work on program understanding into programming exercises that are not completely specified, that do not focus on particular features of the programming language and that involve decisions about the specification and intended behaviour of the program. We are looking at programs that are expected to contain bodged and novel implementations.

We have taken an approach in which we separate problem-solving in the task domain from problem-solving in the programming language. We have based our analysis of the Prolog level on existing synthesis methods used for program design in Prolog, especially programming techniques and data types, so that any critique will be in terms of the design methods that are used by experts and being taught to students.

7.1.1 Programming at the Task Level and Language Level

We have achieved a partial separation of problem-solving in the programming language from problem-solving in the problem domain. We have created an architecture in which decisions in the task domain are represented by task definitions and domain object definitions, and implementation decisions in the programming language are represented by prototype definitions and data type declarations.

7.1.2 Using Synthesis Methods

Our recognition system uses an analysis-by-synthesis approach that is based on software development strategies which have been proposed for use by human programmers (O'Keefe, 1990; Sterling & Lakhotia, 1988). We have automated the process of choosing a task and applying an appropriate programming technique to that task in order to synthesise Prolog code that can be matched to a student's program.

The system synthesises code at run-time, deciding which prototypical Prolog predicates should be used to perform particular tasks. It determines at run-time which prototypes should be applied to which tasks, instead of relying on an explicit declaration. It allows a single prototype to be applied to many tasks in the problem domain, without the system builder having to predict and declare in advance which prototypes can be applied to which tasks.

The behaviour of sub-tasks and prototypes is used to determine whether a prototype could be used to fulfill a particular task. Separating tasks and prototypes is a saving over making an explicit declaration of prototypes for a task (or plans for a goal, in Proust (Johnson, 1990)). Where a group of prototypes uses very similar roles then all the prototypes in that group may be applied to a task, without the need to declare that each individual prototype may be applied to

it. A single declaration of the roles and sub-tasks for a task means that all the plans that use those sub-tasks in those roles may be applied to that task.

7.1.3 Reasoning About Data Objects

We have described design decisions about data structures in both the task domain and the Prolog language domain. A type language and type inference mechanism have been adapted to describe and determine the data structures that students have created. Polymorphic types and a type inference mechanism are suitable to describe many data structure decisions made in Prolog, even though Prolog is not a typed language.

Type inference does not in itself solve the problem of mapping between the students' intentions and implementation in terms of the programming language and their intentions in terms of the problem domain. Some Prolog data types may be used in several different contexts to represent different entities in the problem. The type inference system does not distinguish between these different uses of the same data type. We have devised a separate search-based mechanism to link intentions in the problem domain to their implementation in the programming language.

7.2 Discussion: Implications, Limitations and Extensions

This section explores the implications of our work in terms of some major problems in automated program understanding for intelligent tutoring systems and for software maintenance.

7.2.1 Using Type Inference to Debug Data Structure Decisions

There are two phases in automated debugging. We have focussed on the first phase, in which hypotheses are formed about intentions. We have investigated the requirements of a type inference system for describing intended data structures and making hypotheses about their intended purpose, and for this we have used a mainly descriptive approach. In the second phase code is debugged with respect to those intentions (Murray, 1988). A system that also critiques and debugs type errors can only operate with respect to some prescription about how types should be used. Having identified the intended data types, existing work on type checking could be applied in a further analysis phase in order to identify type errors.

From the viewpoint of implementing type systems, type checking can be seen as merely a degenerate version of type inference in which the inference process has become trivial. But in fact the roles of type checking and type inference are different. Type checking and type inference place a different emphasis on declarations by the programmer. Type checking checks the actual behaviour of the program against a specification of its behaviour. The type declarations that are supplied for a predicate act as an independent specification of the predicate's behaviour, and mismatches between the declarations and the inferred behaviour can easily be detected. Type inference places an emphasis on working out which

types have been used and requires that the minimum of type declarations be supplied by the programmer. Consistency is checked during type inference, but this is consistency with respect to the program (and type inference system) itself rather than consistency with respect to external declarations about the program. The two can usefully be combined, so that a type inference can infer the behaviour of the program independently of type declarations and also check that behaviour against declarations.

Existing research on type inference has focussed on providing support to programmers (Pfenning, 1992). The criteria for type mechanisms are that they must be expressive, reliable and efficient. The type language must be sufficiently expressive to allow the programmers to create and describe the data structures that they need. The process of type inference or type checking must not slow down the the execution of the program. The type checking process must ensure that programs are typed correctly and consistently.

One important criterion for type inference on production software has been that type inference mechanisms must infer only one type description for any variable, and they should not infer more than one correct type description for any part of a program. Maintaining multiple type descriptions and ensuring that type descriptions of sub-programs are combined in a way that is correctly typed quickly becomes a large search problem.

Efficiency is less of a constraint for tutoring systems that for production software. Programming exercises do not typically require long executions or require maximum run-time speed. For an interactive critiquer, a rapid response time is needed from the type system. In our non-interactive system, a rapid response time is not essential. In this context, search-based approaches can be considered.

Ambiguity in type recognition is a problem for tutoring as well as for production software. A small number of alternative interpretations would be tolerable, but if there are a large numbers of different interpretations for each argument then a very large search space would be created. This could be dealt with by including heuristics to reflect which interpretation most closely reflects the student's

intentions, and to enable the system to reject unlikely ones early. But where our understanding of the program is very fragmented, it is difficult to apply such heuristics.

In this situation, we derive a single less informative type description instead of generating many detailed type descriptions. This is similar to the idea of abstraction that is used for recognising code structures in SCENT (Greer & McCalla, 1989). In addition, we have gathered information without using search or problem-specific heuristics within the type inference mechanism. Search and problem-specific heuristics have been used at the domain object level.

7.2.2 Correctness of Synthesised Code

We cannot fully answer the question of whether a particular combination of a prototypical predicate and its sub-goals really will create code that fulfills the task correctly. It is possible to apply a prototypical predicate to some sub-goals of a task and generate code that does not fulfill that task. Proust tackles this problem by insisting that only the named prototypical predicates may be used for a task. So long as the prototypical predicates and sub-goals are specified correctly for a goal, the generated code will be correct. However, this is not a complete solution. All responsibility for ensuring that the synthesised code is correct (or is buggy in some known way) remains with the person who specifies the goals and prototypical predicates. Proust itself cannot validate the plans for a goal or the code that is created from them.

In general it is not possible to decide formally whether a piece of synthesised code is in fact a correct implementation of its specification. This is a major problem in program synthesis. We use data types as one partial way to check that the synthesised code is meaningful.

In our system, code is only synthesised using a prototype if the matches between sub-tasks and prototypical predicates are consistently typed and if they produce predicates that are consistently typed. The data types of the variables for the

predicates are inferred from the combination of the sub-tasks and the prototypical predicate. This could be extended from a general check that the code is consistent to a check on whether it is acceptable code for that task by using expectations about the data types for domain objects manipulated by each task. The set of acceptable types for the arguments to each task could be inferred from the information about the domain objects that are manipulated by each task. This type information could be used to check that the combination of sub-goals and prototypical predicates has produced a task uses the appropriate types. We have not implemented such a check.

The problem of ensuring that synthesised code is correct could be explored further using existing program analysis methods such as dynamic analysis (Looi, 1988b). It would be necessary to apply these methods to the synthesised programs as well as to the student's program. A valuable extension would be to include other behavioural descriptions of goals and their roles within prototypical predicates besides data type descriptions.

7.2.3 Separating Domain–Specific and Language–Specific Knowledge

In our system, domain tasks correspond to Proust's goals and prototypical predicates correspond to Proust's plans. We have weakened the dependence on explicit declaration of the relation between goals and plans. Instead, our system determines which plans are appropriate. Our approach is:

- to use the behaviour of sub-goals and plans to determine whether a plan could fulfill a goal;
- to connect goals to sub-goals directly, so that a plan is chosen if it fulfills the required sub-goals.

In our system, the goal declaration states the sub-goals and the role that each sub-goal must play in any plan that is chosen. A plan is chosen for a goal

if all of the roles in the plan can be fulfilled by sub-goals of the goal used in the same roles. By contrast, a goal declaration in Proust states the plans that may be chosen for that goal. In Proust's structure of goals and plans, goals are linked explicitly to the plans that can be used to implement them, and plans are explicitly linked to the sub-goals that they require. Goals are linked to plans by name and plans are linked to sub-goals by name.

Goals and plans are nevertheless implicitly linked in our system, through the roles that each sub-goal plays in a plan. It is difficult to visualise a role for a sub-goal within a task without also visualising some plan that includes that role. We define a sub-goal for a task and the role that it fulfills only if we have in mind at least one plan that is being used for the task and will make use of the sub-goal in that role. What a system builder gains by specifying roles and sub-goals rather than specifying plans is that if there are other plans that use the same combination of sub-goals in the same roles, then those plans are also applied to that task.

Our prototypical predicates, like Proust's plans, contain elements of the code that is to be matched. Prototypical predicates encapsulate language-specific programming knowledge, whereas the grouping of goals and sub-goals describes the problem independently of the details of its implementation in Prolog.

A prototype need not match all the roles for a goal, so that different prototypes can use different sub-goals and different combinations of sub-goals. Some prototypes require only a subset of the sub-goals that are required by other prototypes. Prototypes which require a subset may be applied to goals for which the larger set of roles is supplied, as well as for goals that have only the smaller set of roles.

The use of goals and sub-goals as a language-independent description could be made more powerful by explicitly grouping together the roles that are expected to appear in the same plan for that goal, and distinguishing these from roles that are expected to appear in some other plans for that goal. It would still be possible to do this without specifying which plans are expected. Then the goal structure

would represent the different abstract breakdowns of the problem into goals and sub-goals, while the prototypical predicates would represent alternative ways to implement the connections between goals and sub-goals.

Indicating which roles are essential to any plan that implements the goal and which are optional would help to prevent inappropriate combinations of goal and plans from being used.

These improvements could be effected by:

- grouping together explicitly the roles in a task that might be expected to be used together in a single realisation;

- indicating explicitly which roles are essential to any such group for a task, and which are optional.

Our system first synthesises code and then tries to match it, whereas Proust matches the elements from the plan directly to the code. Unlike Pascal, Prolog's syntax does not offer easy syntactic markers for matching directly to the templates in the plans. The program syntax is very much altered when the prototypical predicates are combined with tasks. It is easier to match the completed synthesised predicate gainst the student's predicate than it is to isolate syntactic elements from the prototype.

The Proust goal/plan approach does not distinguish between problem solving in terms of the problem domain and problem-solving in terms of the programming language. These are mixed in the definition of goals and plans. In the Programmer's Apprentice, clichés are grouped together but the formalism does not distinguish between domain-specific clichés and language-specific clichés. The model in Proust and the Programmer's Apprentice is of a series of domain plans which are refined until they bottom out at language-specific plans (Johnson, 1990; Rich & Waters, 1990). By contrast, the LISP Tutor, SCENT and ADAPT refine programs through the constructs that they use in the programming language (Anderson *et al*, 1990; Greer *et al*, 1989; Gegg-Harrison, 1992). Our system has

a separate structure for each of these: a task structure for domain plans which maps to a prototype structure for programming techniques.

Our division between the task structure and the prototype structure is somewhat overloaded. Tasks are supposed to represent *what* needs to be done, prototypes *how* it should be done. The task structure also forms a breakdown of the task in terms of the problem domain (or at least, independent of the programming language), whereas the prototypes represent techniques that are specific to the programming language. These two distinctions can be drawn in other ways, for instance there may be tasks that appear in many problem domains.

Just as the choice of plans to fulfill a particular goal affects the creation of sub-goals, so the choice of technique to fulfill a particular task affects the choice of sub-tasks. The form of the task/sub-task structure in the student's program is dependent on the choice of techniques. So tasks and techniques are mutually dependent, and the tasks and sub-tasks do not fully distinguish between a description of the solution in terms of the problem domain and a description of the solution in terms of programming techniques.

A complete separation between goals and plans, or between tasks and techniques, is not possible, but a clearer distinction between tasks and programming techniques would be achievable. At present, plans for solving the problem in terms of the problem domain are linked to a separate structure for solving the problem in terms of the implementation language. In future, the implementation structure could itself be a complex plan structure in which prototypes may be explicitly related to one another. A prototype may be expressed as a refinement of another prototype or as embodying a variant of similar programming techniques. We have already taken some steps in this direction by including join specifications which embody the fact that one prototype is the clausal join of two component prototypes.

7.2.4 Programming Techniques

Prototypical predicates encapsulate knowledge about programming techniques. Prototypical predicates are related to each other in terms of which programming techniques they use. Some prototypical predicates may be seen as arising from common programming schemas. For both tutorial and recognition purposes, explicit information is needed about which techniques are represented by the prototypes.

In the present system, each prototypical predicate is represented and treated as an independent entity, unconnected with any other prototypical predicate.

Prototypical predicates could be linked together in terms of the programming constructs that they use. These connections could be in terms of SCENT's abstraction or aggregation hierarchies (Greer & McCalla, 1989), or they could be in terms of the schema hierarchies proposed by Gegg-Harrison (Gegg-Harrison, 1992). The exact connections that are required for a Prolog tutoring system remain to be explored. Plans can be connected together if they are created from common abstract schemas. Alternatively, they can also be connected together if they are elaborations of the same simpler program.

A connection in terms of Gegg-Harrison's schemas was considered but was not used for this problem. Our investigation of student's programs showed that the Prolog predicates covered by Gegg-Harrison's schemas accounted for only a very small part of the students' programs for the noughts and crosses exercise. Large parts of the exercise used non-recursive predicates and recursions that did not correspond to Gegg-Harrison's schemas. The schema language must also describe other techniques such as failure-driven loops and case analyses (Brna *et al*, 1991).

One of the real difficulties is to link programming schemas or prototypes in the object language with their purpose. In the context of a particular programming task, when should a particular technique be chosen? A "techniques editor" provides a range of abstract programming techniques that can be selected, combined

and completed by the programmer (Bowles & Brna, 1993). Even a techniques editor does not necessarily provide any clues about when a particular programming technique should be used in a particular program. Some link between programming techniques and the tasks for which they are suitable is needed. Techniques must be described in terms of abstract properties which can be related to abstract properties of the task for which they are being performed.

Particular techniques should be used, and that they should be used in particular ways. There is a notion of the correct or incorrect use of a technique, of a programming technique that has been applied correctly versus one that has been applied incorrectly or mutated beyond recognition. This is dependent on what the students have been taught. The limitations of this are evident in parts of the thesis. Some notion of the correct (or at least conventional) use of data types is implicit in the type inference mechanism.

The difficulty of recognising programming techniques in programs derives from the following problems:

- It may be difficult to specify the technique formally.
- If a technique cannot be specified formally but can only be seen as an informal pattern or hint that is used in the design of a program in an unrestricted way, then that technique may be mutated beyond recognition when it appears in program.
- When dealing with novice programs, techniques that are typically used in particular ways may be mutated into other forms that are unique and difficult to recognise.

7.2.5 Reconstructing Decisions

We have tried to obtain as much information as possible from the program itself, and have not used any information about the program from other sources.

Even in a highly interactive system that explicitly questions the user about the user's intentions at many levels, there will still be gaps. There may be differences between what the user has specified at one level and what the user has done at a lower level. The users may have difficulty encoding their intentions in the languages that are provided, so the intentions that the user has conveyed to the system may not be the intentions that the user actually has. An analysis of the actual structure of the program and its behaviour independent of the user's stated intentions is needed in order to detect mismatches between the user's real intentions and what the user has described to the system.

A completely automated system cannot reliably recover the programmer's intentions in the problem domain only from the program itself. The purpose of a piece of code and the reasons why the code was written in a particular way are "compiled out" of the code itself. The problem is comparable to the Guidon project's attempt to reconstruct medical knowledge from the Mycin medical expert system. This project showed that information that is not expressed explicitly could not be be recovered from the code alone (Clancey, 1987). Various kinds of information has been compiled out by the system builders, in particular knowledge about the problem domain which justified individual rules (Swartout, 1983), and control knowledge about how information should be used within a diagnostic inference strategy (Hasling *et al*, 1983; Chandrasekaran *et al*, 1989). In order to understand Mycin's design, the medical domain knowledge and diagnostic strategy had to be obtained from elsewhere.

Other sources of information about the program include:

- maintaining a history of the development of the code (Bowles & Brna, 1993);
- analysing free–text comments (Biggerstaff, 1989);
- maintaining information about different aspects of the design and their links to the actual implementation (Biggerstaff, 1989; Iscoe, 1992);

- querying the programmer about aspects of the code and design after the code has been developed (Fuchs & Fromherz, 1992).

The approach of maintaining a machine-readable history of the operations by which the code was created has been taken in tutoring systems (Bonar & Cunningham, 1988) and in editors for novices (Bowles & Brna, 1993; Bowles *et al*, 1993; Vargas-Vera *et al*, 1993). This approach has also been proposed for the cliché-based approach to software design in the Programmer's Apprentice project (Rich & Waters, 1990), but mechanisms for maintaining information about how code is derived from clichés have not been implemented (Wills, 1990).

Information from a human systems analyst can be combined with machine assistance to reconstruct the design. A programmer can supply information about which clichés have been used in a program. For example the programmer supplies this information to a system which uses knowledge about clichés to optimise the program automatically (Fuchs & Fromherz, 1992).

Our approach might usefully be combined with these various other sources of information in an interactive system. Such a system might maintain a history of the student's activities so far and query the student about intentions. It might ask the user to supply descriptions of domain objects or type declarations and then compare the inferred declarations with the ones the user has implemented. Having made tentative inferences about design decisions, it might then hold a tutorial dialogue with the user in which students describe their intentions and the system proposes improvements to their code.

7.2.6 Scale

One important issue in tutoring systems for computer programming is the large amount of information that must be supplied by hand to the system in order to describe the possible implementations of each task. Apropos and Talus used a different reference program for each implementation, correct and incorrect

(Looi, 1988b; Murray, 1988). For example, even in as small a program as reversing a list, reference programs are supplied to Apropos for three distinct implementations of reversing a list, together with buggy clauses for those implementations and a reference program for the sub–task of appending two lists. It is unlikely ever to be practical to supply so much detail for all the unique parts and variants of a much larger program.

Conversely, Will's recogniser deals with much larger programs but it does not try to recognise buggy implementations, and it has only been run on two programs which were extensively studied and for which the clichés were derived (Wills, 1990). A question remains whether it feasible to specify all the parts and all the variants of a large program at such a fine level of detail.

Gegg–Harrison has shown that for a limited class of recursive programs, many variants of a program, even variants that use different programming techniques, can be generated automatically from a single reference predicate (Gegg-Harrison, 1992). It remains to be seen how far this can be extended to deal with programs outside this class and with buggy programs.

7.3 Applications and Further Work

In this section we consider the applications of our work and extensions to our work that are most likely to lead to useful results. We first summarise briefly the extensions to the system that would improve its behaviour as a recogniser. We next describe the implications of using type inference in tutoring systems for computer programming.

7.3.1 Extensions to the System

This section describes some extensions to the system that would improve its behaviour as a recogniser and would make it a more useful teaching tool.

Extending the Library

The library of tasks, prototypes and domain objects that we have developed for demonstration purposes would be incomplete for a realistic analysis. Even for the single task of finding an empty square described in Section 6.6, the range of tasks, prototypes and domain objects must be extended if the system is to be effective. Further study of noughts and crosses programs is needed to derive a representative set of approaches that students use for all the tasks in noughts and crosses. This would mean studying a larger set of programs and more tasks within those programs.

The system might also be applied to exercises other than the noughts and crosses program. The same data type declarations would be used, but extensions to the library of prototype definitions would be required, and replacement of task and domain object definitions. Different implementations of the new exercise would need to be studied.

Representing Programming Techniques

As described in Sections 7.2.3 and 7.2.4, explicit links between prototypes could usefully be provided to improve both recognition and critiquing.

A formal classification of programming techniques (such as that provided by (Gegg-Harrison, 1991)) could be linked to the prototypes and used for tutoring. The system could produce more useful commentary from this explicit information about the programming techniques that the prototypes represent. At present, the names of programming techniques offer a clue to the programming techniques that they embody, and the join specifications link composed prototypes to smaller prototypes that represent individual programming techniques.

Representing Roles

In the system at present, a role description in a task definition is linked to a role slot in a prototypical clause explicitly by name and implicitly through a check

that the correct data types are manipulated. The names of roles must therefore embody almost all the information about the role of the sub-task within the prototype. Ideally we would like to extend the idea of roles to include more descriptive properties, so that a role specification would fill a slot in a prototype if it had the appropriate properties.

Dealing with Variations

Our work is limited by an inability to reason about equivalence of programs. The system's recognition of code structure is very fragile, compared to GRASPR (which abstracts to data flow and control flow), Talus (which uses heuristics and proves equivalence of programs), Apropos (which uses heuristics and dynamic analysis) or Proust (which uses heuristics). These different approaches could be applied to matching in our system.

This would open up further questions about control. In the programs which we analyse there is a large number of tasks, there are many ways in which the problem can be broken down and there is a large range of possible implementations. These all add up to a large search problem, which is further complicated by the possibility of incorrect and novel implementations. If these approaches are used, they must be integrated with the recognition of the task structure, which may be affected by transformations to the code. As a first step, the transformation operations that we propose in Section 5.4 could be integrated into the control and applied in some standard way. Further work will be needed to find ways to control the search that may result.

7.3.2 The Application of Type Inference to Tutoring Programming

In this section, we consider in a broad way the potential role of type inference in teaching computer programming. Our work on inferring intended types can be applied to intention-based assistance for debugging of type errors in typed lan-

guages. In typed languages, the programmer presents detailed type information for each predicate. However, novices may experience difficulty in creating and using properly typed predicates, and in debugging the type errors that are reported by such systems (Soosaipillai, 1990). For these systems, a descriptive and intention–based approach that can compare the types actually produced with the types as described may help the student to pin–point and understand type errors. We describe possible extensions to the type language and a potential application of type inference to misconception modelling.

Type Languages for Intention–Based Debugging

We have identified the need to represent inclusion and parametric polymorphism in order to represent the kinds of data structures that students create (Section 4.11). It is not yet feasible to perform type inference on inclusion (i.e. sub–types) and parametric polymorphism simultaneously, and so we used anonymous types instead of inclusion. It remains to be investigated whether these two requirements are sufficient, or whether students' programs should also be investigated against type languages with other properties. A use can be envisaged for other descriptive criteria, such as whether a list is sorted. Such criteria are incorporated in some proposed type languages for Prolog, e.g (Dietrich & Hagl, 1988). They would require integrating different approaches to inference, such as dynamic analysis.

We chose data type languages that describe the data structures in terms of the Prolog implementation rather than in terms of the problem domain. When programmers working on a project agree on naming conventions for functors that reflect the problem domain, it would be possible to make more use of functor names to infer some domain intentions within the type inference mechanism itself. We found that this did not work for students' programs, because they are not usually required to follow naming conventions in their programming exercises, but in large–scale real–life projects this approach would be more feasible (Biggerstaff, 1989).

Modelling Students' Misconceptions about Data Types

Students' misconceptions about types and type mechanisms can be modeled by analysing their programs in terms of different type mechanisms. This has been done for modelling students' misconceptions about the Prolog interpreter itself in terms of variants of the actual interpreter (Fung *et al*, 1990). Different misconceptions can surface in different parts of the program, and this can be modelled by applying different mechanisms to different parts of the problem. One mechanism can be applied when another has failed.

7.4 Summary

We have developed an architecture to deal with some of the problems of reconstructing student's design decisions in the programming exercises that occur in the later stages of learning to program. The problems encountered in these exercises are:

- decisions about data representations are not part of the problem specification;
- problem solving is required in terms of both the chosen programming language and a more abstract domain of general problem-solving and problem-solving in the domain of the exercise;
- the design of the program is made up of a large number of interacting decisions.

Our solution to this entails:

- reasoning explicitly about the objects that are represented by data structures and their intended representations;
- distinguishing between intentions in the programming language and intentions in the problem domain;
- using synthesis methods to isolate meaningful fragments of the program and then driving the recognition process from these fragments.

In particular, the system provides an advance in four different areas. First, polymorphic type inference has been adapted to serve multiple needs related to the recognition of data structure decisions at the language–specific implementation level. The choice of data structures and the way in which they are represented are significant decisions which have important effects on the design of the rest of the system. In order to understand and critique student's programs, we need to recognise them. Our solution has provided support for helping the user understand and critique student's programs.

Second, decisions about data structures are represented in terms of the problem domain and the programming language. Abstract descriptions of data types are linked to the objects in the problem domain that they are intended to represent. A search–based mechanism connects data structures with their meanings in the problem domain.

Third, decisions about code structure are represented in terms of the problem domain and the programming language. Decisions in the problem domain are represented by a task structure and implementation decisions in the programming language are represented by prototypes. The relation between these decisions is used to drive an analysis–by–synthesis process in which prototypes are applied to particular tasks only when they are likely to be appropriate. The system uses an explicit representation of roles with tasks and prototypes to determine when a prototype is appropriate, and it uses type inference to check that the synthesised code for the task is indeed correct.

Fourth, we have successfully developed a technique, clausal split, which can be used to identify the components of a Prolog program when their combination cannot be recognised. It can be used to isolate incorrect and unfamiliar implementations and to identify correct components of these implementations.

This system is evidence that through a combination of techniques some progress can be made to support critiquing. It remains to be seen whether real progress can be attained through the integration of other techniques as outlined in this thesis. The system has been applied to intermediate students' programs and

found to perform at a reasonable level. A fully automated intention-based critique of intermediate students' programs remains remote, but by integrating the techniques we have described in this thesis with tools that capture some of the students' design decisions as they are made, we may hope to develop usable tools for teaching intermediate computer programming.

References

Adam, A. and Laurent, J. (1980). LAURA: A system to debug student programs. *Artificial Intelligence*, 15:75–122.

Adelson, B. and Soloway, E. (1985). The role of domain experience in software design. *IEEE Transactions on Software Engineering*, 11:1351–1360.

Adelson, B., Littman, D., Ehrlich, K., Black, J. and Soloway, D. (1985). Novice–expert differences in software design. In *Human–Computer Interaction — INTERACT 84*, pages 473–478. Elsevier.

Anderson, J. R. and Skwarecki, E. (1986). The automated tutoring of introductory computer programming. *Communications of the ACM*, 29(9):842–849.

Anderson, J. R., Farrell, R. and Sauers, R. (1984). Learning to program in LISP. *Cognitive Science*, 8:87–129.

Anderson, J. R., Boyle, C. F. and Yost, G. (1985). The geometry tutor. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1–7.

Anderson, J. R., Boyle, C. F., Corbett, A. T. and Lewis, M. W. (1990). Cognitive modeling and intelligent tutoring. *Artificial Intelligence*, 42:7–50.

Barr, A., Beard, M. and Atkinson, R. (1976). The computer as a tutorial laboratory. *International Journal of Man-Machine Studies*, 8:567–596.

Bental, D. (1992). Using clausal join and clausal split to recognise language-specific programming design decisions. In *Proceedings of the First Workshop on Artificial Intelligence and Automated Program Understanding*, pages 37–41. AAAI.

Biggerstaff, T. J. (1989). Design recovery for maintenance and re-use. *IEEE Computer*, pages 36–49.

Bonar, J. and Cunningham, R. (1988). Bridge: An intelligent tutor for thinking about programming. In Self, J., (ed.), *Artificial Intelligence and Human Learning: Intelligent Computer-Aided Instruction*, chapter 24. Chapman and Hall.

Bowles, A. and Brna, P. (1993). The notion of programming plans. In Brna, P., Ohlsson, Stellan and Pain, Helen, (eds.), *Proceedings of the World Conference on Artificial Intelligence and Education*, pages 378–385. AACE.

Bowles, A., Roberston, D., Vasconcelos, W., Vargas-Vera, M. and Bental, D. (1993). Applying Prolog programming techniques. Technical report, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, UK, Research Paper 641.

Brecht, B. and Jones, M. (1988). Student models: The genetic graph approach. *International Journal of Man-Machine Studies*, 28:483–504.

Brna, P., Bundy, A., Dodd, A., Eisentadt, M., Looi, C.-K., Pain, H., Robertson, D., Smith, B. and van Someren, M. (1991). Prolog programming techniques. *Instructional Science*, 20(2):111–133.

Brown, D. C. and Chandrasekaran, B. (1989). *Design Problem Solving: Knowledge Structures and Control Strategies.* Pitman.

Brown, J.S., Burton, R.R. and de Kleer, J. (1981). Pedagogical, natural language and knowledge engineering techniques in SOPHIE I, II and III. In Sleeman, D. and Brown, J.S., (eds.), *Intelligent Tutoring Systems*, pages 227–282. Academic Press.

Bundy, A. (1988). Proposal for a recursive techniques editor for Prolog. Technical Report 394, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, UK.

Bundy, A., Pain, H., Brna, P. and Lynch, L. (1986). A proposed Prolog story. Technical Report 283, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, UK.

Bundy, A., Grosse, G. and Brna, P. (1991). A recursive techniques editor for Prolog. *Instructional Science*, 20:135–172.

Burton, R. R. and Brown, J. S. (1981). An investigation of computer coaching for informal learning activities. In Sleeman, D. and Brown, J. S., (eds.), *Intelligent Tutoring Systems*, pages 79–98. Academic Press.

Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522.

Carroll, J. M. and Aaronson, A. P. (1988). Learning by doing with simulated intelligent help. *Communications of the ACM*, 31(9):1064–1079.

Chandrasekaran, B., Tanner, M. C. and Josephson, J. R. (1989). Explaining control strategies in problem solving. *IEEE Expert*, 4:9–24.

Clancey, W. J. (1987). *Knowledge-Based Tutoring: The GUIDON Program.* MIT Press.

Corbett, A. T., Anderson, J. R. and Patterson, E. G. (1988). Student modeling and tutoring flexibility in the Lisp intelligent tutoring system. In *Proceedings of the International Conference on Intelligent Tutoring Systems*, pages 83–106. Ablex.

Corbett, A. T., Anderson, J. R. and O'Brien, A. T. (1993). The predictive validity of student modelling in the ACT programming tutor. In Brna, P., Ohlsson, Stellan and Pain, Helen, (eds.), *Proceedings of the World Conference on Artificial Intelligence and Education*, pages 457–464. AACE.

Dietrich, R. and Hagl, F. (1988). A polymorphic type system with subtypes for Prolog. In Ganzinger, H., (ed.), *Proceedings of the Second European Symposium on Programming*, pages 79–93. Springer-Verlag.

Eisenstadt, M. and Brayshaw, M. (1986). The Transparent Prolog Machine (TPM): An execution model and graphical debugger for logic programming. Technical Report 21, Human Cognition Research Laboratory, The Open University, Milton Keynes, UK.

Elsom-Cook, M. and du Boulay, B. (1988). The requirements of conceptual modelling systems. In Self, J., (ed.), *Artificial Intelligence and Human Learning: Intelligent Computer-Aided Instruction*, chapter 22, pages 361–373. Chapman and Hall.

Fischer, G. (1987). A critic for LISP. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 177–184.

Foss, C. L. (1987). Learning from errors in AlgebraLand. IRL Tech. Report 3, Institute for Research on Learning, Xerox Palo Alto Research Center, Ca.

Frühwirth, T. W. (1988). Type inference by program transformation and partial evaluation. In *Proceedings of the Workshop on Meta-Programming in Logic Programming*, pages 199–212, Bristol, UK.

Fuchs, N. E. and Fromherz, M. P. J. (1992). Schema-based transformations of logic programs. In Clement, T. P. and Lau, K.-K., (eds.), *Logic Program Synthesis and Transformation, Workshops in Computing*. Springer Verlag.

Fuh, Y.-C. and Mishra, P. (1987). Type inference with subtypes. Technical Report Technical Reprot 87-25, SUNY at Stony Brook.

Fung, P., Brayshaw, M., Du Boulay, B. and Elsom-Cook, M. (1990). Towards a taxonomy of novices' misconceptions of the Prolog interpreter. *Instructional Science*, 19(4/5):311–336.

Gegg-Harrison, T. S. (1989). Basic Prolog schemata. Technical Report CS-1989-20, Department of Computer Science, Duke University, Durham, North Carolina.

Gegg-Harrison, T. S. (1991). Learning Prolog in a schema-based environment. *Instructional Science*, 20:173–192.

Gegg-Harrison, T. S. (1992). ADAPT: Automated debugging in an adaptive Prolog tutor. In Frasson, C., Gauthier, G. and McCalla, G.I., (eds.), *Second International Conference on Intelligent Tutoring Systems*, pages 343–350. Springer-Verlag.

Greer, J. E. and McCalla, G. I. (1989). A computational framework for granularity and its application to educational diagnosis. In *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 1, pages 477–482.

Greer, J. E., Mark, M. A. and McCalla, G. I. (1989). Incorporating granularity-based recognition into SCENT. In Bierman, Breuker and Sandberg, (eds.), *Proceedings of the 4th International Conference on AI and Education*, pages 107–115. IOS.

Hasling, D. W., Clancey, W. J. and Rennels, G. (1983). Strategic explanations for a diagnostic consultation system. *International Journal of Man-Machine Studies.*

Hill, P. M. and Topor, R. W. (1992). A semantics for typed logic programs. In Pfenning, F., (ed.), *Types in Logic Programming*, pages 1–62. MIT Press.

Iscoe, N. (1992). Program understanding: Using and gathering application domain knowledge. In *Proceedings of the First Workshop on Artifical Intelligence and Automated Program Understanding*, pages 70–71. AAAI.

Johnson, W. L. and Soloway, E. (1984). Intention-based diagnosis of programming errors. In *Proceedings of the National Conference on Artificial Intelligence*, pages 162–168.

Johnson, W. L. (1990). Understanding and debugging novice programs. *Artificial Intelligence*, 42:51–97.

Joni, S.-N. A. and Soloway, E. (1986). But my program runs! Discourse rules for novice programmers. *Journal of Educational Computing Research*, 2:95–125.

Kant, E. (1985). Understanding and automating algorithm design. *IEEE Transactions on Software Engineering*, 11:1361–1373.

Kowalski, R. (1979). Algorithm = logic + control. *Communications of the ACM*, 22:424–436.

Lakhotia, A. and Sterling, L. S. (1987). Composing recursive logic programs with clausal join. In *Proceedings of the Workshop on Partial and Mixed Computation.*

Letovsky, S. I. (1988). *Plan Analysis of Programs.* Unpublished Ph.D. thesis, Department of Computer Science, Yale University, New Haven, Ct.

Looi, C.-K. (1988a). Apropos2: A program analyser for a Prolog intelligent teaching system. In *Proceedings of the International Conference on Intelligent Tutoring Systems.*

Looi, C.-K. (1988b). *Automatic Program Analysis in a Prolog Intelligent Teaching System.* Unpublished Ph.D. thesis, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, UK.

Lutz, R. (1991). Plan diagrams as the basis for understanding and debugging Pascal programs. In Eisenstadt, M., Keane, M. T. and Rajan, T., (eds.), *Novice Programming Environments*, chapter 11, pages 243–285. Lawrence Erlbaum Associates.

Lutz, R. (1993). *Towards an Intelligent Debugging System for Pascal Programs: On the Theory and Algorithms of Plan Recognition in Rich's Plan Calculus.* Unpublished Ph.D. thesis, The Open University, Milton Keyenes, UK.

Macmillan, S. A. and Sleeman, D. H. (1987). An architecture for a self-improving instructional planner for intelligent tutoring systems. *Computational Intelligence*, 3:17–27.

McCalla, G. I. and Greer, J. E. (1988). SCENT-3: An architecture for intelligent advising in problem-solving domains. In *Proceedings of the International Conference on Intelligent Tutoring Systems*, pages 140–161. Ablex.

McCalla, G. I., Bunt, R. B. and Harms, J. J. (1986). The design of the SCENT automated adviser. *Computational Intelligence*, 2:76–92.

Miller, M. L. and Goldstein, I. P. (1977). Structured planning and debugging. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 773–779.

Miller, M. L. (1981). A structured planning and debugging environment for elementary programming. In Sleeman, D. and Brown, J.S., (eds.), *Intelligent Tutoring Systems*, pages 119–135. Academic Press.

Miller, P. L. (1984). *A Critiquing Approach to Expert Computer Advice: Attending.* Pitman.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375.

Mishra, P. (1984). Towards a theory of types in Prolog. In *Proceedings of the 1984 International Symposium on Logic Programming*, pages 289–298. IEEE Society Press.

Moore, J. L. and Sleeman, D. (1988). Enhancing PIXIES's tutoring capabilities. *International Journal of Man-Machine Studies*, 28:605–623.

Murray, W. R. (1987). Automatic program debugging for intelligent tutoring systems. *Computational Intelligence*, 3:1–16.

Murray, W. R. (1988). *Automatic Program Debugging for Intelligent Tutoring Systems.* Pitman.

Murray, W. R. (1989). Control for intelligent tutoring systems: A blackboard-based dynamic instructional planner. In Bierman, Breuker and Sandberg, (eds.), *Proceedings of the 4th International Conference on AI and Education*, pages 150–168. IOS.

Mycroft, A. and O'Keefe, R. A. (1984). A polymorphic type system for Prolog. *Artificial Intelligence*, 23:195–307.

O'Keefe, R. A. (1990). *The Craft of Prolog.* MIT Press.

Pfenning, F., (ed.). (1992). *Types in Logic Programming.* MIT Press.

Reiser, B. J., Anderson, J. R. and Farrell, R. G. (1985). Dynamic student modelling in an intelligent tutor for LISP programming. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 8–14.

Reiser, B. J., Ranney, M., Lovett, M. C. and Kimberg, D. Y. (1989). Facilitating students' reasoning with causal explanations and visual representations. In Bierman, Breuker and Sandberg, (eds.), *Proceedings of the 4th International Conference on AI and Education*, pages 228–235. IOS.

Rich, C. and Waters, R. C. (1983). The programmer's apprentice project: a research overview. Technical report, MIT, Cambridge, Mass, AI Memo 1004.

Rich, C. and Waters, R. C. (1990). *The Programmer's Apprentice.* ACM Press, New York.

Rist, R. S. (1991). Knowledge creation and retrieval in program design: A comparison of novice and intermediate programmers. *Human-Computer Interaction*, 6:1–46.

Sack, W. (1990). Finding errors by overlooking them. In Frasson, C. and Gauthier, G., (eds.), *Intelligent Tutoring Systems: At the Crossroad of Artificial Intelligence and Education*, pages 206–233. Ablex.

Shapiro, E. (1983). *Algorithmic Prgram Debugging.* MIT Press.

Simon, H. A. (1973). The structure of ill structured problems. *Artificial Intelligence*, 4:181–201.

Smolka, G. (1987). Tel (version 0.9) report and user manual. Technical report, Universität Kaiserslautern, Germany, SEKI Report SR-87-11.

Soosaipillai, H. (1990). An explanation based polymorphic type checker for standard ml. Technical Report MSC Dissertation, Department of Computer Science, Heriot-Watt University, Edinburgh, UK.

Spohrer, J. G. and Soloway, E. (1986). Analysing the high frequency bugs in novice programs. In Soloway, E. and Iyengar, S., (eds.), *Empirical Studies of Programmers*, pages 230–251. Ablex.

Spohrer, J. C. and Soloway, E. (1989). Simulating student programmers. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 543–549.

Spohrer, J. C., Soloway, E. and Pope, E. (1985). A goal/plan analysis of buggy Pascal programs. Technical Report YALEU/CSD/RR 392, Yale University Department of Computer Science, New Haven, Ct.

Steier, D. M. and Kant, E. (1985). The roles of execution and analysis in algorithm design. *IEEE Transactions on Software Engineering*, 11:1375–1386.

Sterling, L. S. and Lakhotia, A. (1988). Composing Prolog meta-interpreters. In Bowen, K. A. and Kowalski, R. A., (eds.), *Proceedings of the Fifth International Conference and Symposium on Logic Programming.* MIT Press.

Sterling, L. S. and Shapiro, E. (1986). *The Art of Prolog: Advanced Programming Techniques.* MIT Press.

Stevens, A., Collins, A. and Goldin, S. E. (1981). Misconceptions in students' understanding. In Sleeman, D. and Brown, J.S., (eds.), *Intelligent Tutoring Systems*, pages 13–24. Academic Press.

Swartout, W. R. (1983). XPLAIN: A system for creating and explaining expert consulting programs. *Artificial Intelligence*, 21:285–325.

Tamaki, H. and Sato, T. (1984). Unfold/fold transformations of logic programs. In *Proceedings of the Second International Conference on Logic Programming*, pages 127–138.

Taylor, J. (1990). Analysing novices analysing Prolog: What stories do novices tell themselves about Prolog? *Instructional Science*, 19:283–309.

van Someren, M. W. (1990). What's wrong? understanding beginners' problems with Prolog. *Instructional Science*, 19:257–282.

Vargas-Vera, M., Vasconcelos, W. and Robertson, D. (1993). Building large-scale Prolog programs using a techniques editing system. In *International Logic Programming Symposium.* MIT Press.

Visser, W. and Hoc, J.-M. (1990). Expert software design strategies. In Hoc, J.-M., Green, T. R. G., Samurcay, R. and Gilmore, D. J., (eds.), *Psychology of Programming*, pages 235–249. Academic Press.

Waters, R. C. (1988). Program translation via abstraction and reimplementation. *IEEE Transactions on Software Engineering*, 14(8):1207–1228.

Wenger, E. (1987). *Artificial Intelligence and Tutoring Systems.* Morgan Kaufmann.

Wertz, H. (1982). Stereotyped program debugging: An aid for novice programmers. *International Journal of Man-Machine Studies*, 16:379–392.

White, R. (1988). Effects of Pascal knowledge on novice Prolog programmers. Technical Report 399, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, UK.

Williams, B. C. (1984). Qualitative analysis of MOS circuits. *Artificial Intelligence*, 24.

Wills, L. M. (1990). Automated program recognition: A feasibility demonstration. *Artificial Intelligence*, 45:113–171.

Wills, L. M. (1992). Automated program recognition by graph parsing. Technical Report Technical Report 1358, MIT Artifical Intelligence Laboratory, Cambrdige, Mass.

Xu, J. and Warren, D. S. (1988). A type inference system for Prolog. In Bowen, K. A. and Kowalski, R. A., (eds.), *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 604–619. MIT Press.

Yardeni, E., Frühwirth, T. W. and Shapiro, E. (1992). Polymorphically typed logic programs. In Pfenning, F., (ed.), *Types in Logic Programming*, pages 63–90. MIT Press.

Appendix A

Examples of Design Errors

We give a detailed analysis of examples of design errors that we have found in the student's programs. The examples are classified according to the model of design skills that we presented in Chapter 3.

We have aimed to identify, describe and classify as many design errors as possible in these programs. In our study worked entirely from the completed code and the comments in the code, and we did not observe students while they wrote their programs. We cannot determine the exact causal sequence of design errors, and many different sequences can be imagined. As a result, classification is based partly on subjective interpretation of how the error arose, and more than one interpretation is possible for some errors. Students were asked to provide comments in their code that described their design decisions, and those comments suggested explanations for some of the errors that arose.

We use the following classification scheme, in which the top level describes the problem space being explored, and the lower level describes the manifestations we found of difficulties in that problem space.

1. General design: structure of a solution
    - Levels of abstraction
    - Scattering tasks through the code
    - Repetition of code
    - Repetition of execution

- Code that is never called
- Failing to validate an input argument
- Mis-use of data constructors

2. Designing in an unfamiliar problem domain
   - Failure to realise all constraints
   - Failure to realise interactions
3. Design in an unfamiliar language
   (a) Misconceptions about Prolog primitives
      - Primitives
      - Analogy

   (b) Misconceptions about Prolog techniques
      - List recursions
      - Case analyses
      - Loops to validate inputs

   (c) Choice of data structures

A.1 General Design: Structure of a solution

A.1.1 Levels of abstraction

P3 P3's code for detecting the end of the game fails to distinguish between two different cases. The code looks like a recursion with two base cases, but it is no such thing. Actually it has two separate cases, which ought to be separate sub-procedures. The first case is a recursion in the first two clauses. Here the code is looking for a line by some player. Only if this recursion fails altogether is the last clause tried to look for a draw.

```
end_state([Line|_],Player) :-
        is_line(Line, Player).
end_state([_|B],Player) :-
        end_state(B,Player),!.
end_state([line(A,B,C),line(D,E,F),line(G,H,I),_,_,_,_,_],_):-
        nonvar(A), nonvar(B), nonvar(C),
        nonvar(D), nonvar(E), nonvar(F),
        nonvar(G), nonvar(H), nonvar(I).
```

The structure of P3's code does not distinguish clearly between looking for a win by some player and looking for a draw. The standard structure would be to put the case analysis at the top level and the recursion at a lower level, roughly as follows:

```
end_state(State, Player) :- win(State, Player) :- !.
end_state(State, Player) :- draw(State, Player).

win([Line|_],Player) :-
        is_line(Line, Player).
win([_|B],Player) :-
        win(B,Player),!.

draw([line(A,B,C),line(D,E,F),line(G,H,I),_,_,_,_,_,_],_):-
     nonvar(A), nonvar(B), nonvar(C),
     nonvar(D), nonvar(E), nonvar(F),
     nonvar(G), nonvar(H), nonvar(I).
```

Effect: readability, verification.

P4 A typical problem was the use of integer identifiers for squares without any relation that grouped particular squares together. P4's `choose_move/3` code explicitly uses square numbers leaving implicit the reasons for choosing those squares. The code does not refer explicitly to the centre, corner or sides, but uses their numeric representations. Here, the computer is intended to pick any corner:

```
choose_move(state(..),Player,move(Player,4)):- ...
```

To understand this, it is necessary to know that corners are numbered evenly in the magic squares representation used by P4. This code also hides the fact that *any* of the four corners, not just 4, would do as well. In Prolog, it is usual to name important properties and relations by creating predicates for them, thus:

```
choose_move(state(..),Player,move(Player, Square)):-
        ..., corner(Square).

corner(4).
```

P4's program does in fact use the explicit relations **corner** and **centre** elsewhere. Here is the code implicitly finding an **opposite corner**:

```
choose_move(state( .. Corner .. ),Player,move(Player,Opposite)):-
        corner(Corner),!,
        Opposite is 10-Corner.

corner(8).
corner(4).
corner(6).
corner(2).
```

The **corner** is explicit but the **opposite corner** is implicit in the subtraction. P4 has done a reasonable job of making the intention of the code clear by using meaningful variable names, but it is preferable explicitly to create a separate predicate which computes an opposite corner.

P4 P4's implementation of `choose_move/3` code does not reflect the intentions of the programmer as stated. There are several levels of bug going on here.

At the lowest level, one of the clauses in `choose_move/3` has an extra argument which means that it is never called. This clause is intended to block any attempts at a fork.

However, this is not a major disaster. The program's default behaviour also blocks some though not all forks.

More surprisingly, the procedure as implemented would permit or even force some very simple forks if it were run. The algorithm for making a fork requires that one takes a square that is shared by two lines in which the player has one token. This program tries use the same method to block a fork, that is, it would take a square that is shared by two lines in which one's opponent has squares. This does not work for blocking a fork, indeed there are cases where it forces ones opponent to fork. Instead one must force the opponent to move their token somewhere that will not give them a fork, a more complex bit of reasoning.

It is not obvious on reading or running the program that the code for blocking a fork cannot be called. The reasons are:

1. Some forks are indeed blocked by the default.

2. The concept of blocking a fork is not given an explicit procedure name. It is merely one of several `choose_move/3` clauses. Thus there is no failure to call an appropriate procedure, which would have made the error obvious in the tracer.

3. `choose_move/3` is written as a long sequence of clauses each of which contains a complex pattern in the head. It is hard to spot that one pattern contains an extra argument and so it will never match any call.

Effect: the program does not match its documentation.

P4 P4's `choose_move/3` code has several examples of relations between data that have been compiled out.

```
choose_move(state([My_last_move|My_former_moves],
                  _YourMoves,
                  Blanks),
            Player,
            move(Place, Player)
        ):-
   member(My_other_sq,My_former_moves),
   member(Place, Blanks),
   line(My_last_move,My_other_sq, Place),!.
```

Here are some deeply implicit game-playing concepts. The object is to find a pair, that is a line in which I have two squares and there is one empty square. The notion of a "pair" is not made explicit. The calls to `member/2` assume that the underlying representation will always be lists of positions.

Another clause in this program implements the notion of a "fork":

```
choose_move(state(_,Mymoves,_,Blanks,_),Player,move(Player,Z)):-
   member(X1,Mymoves),
   member(X2,Mymoves),
   X1 \== X2,
   member(Z1,Blanks),
```

```
member(Z2,Blanks),
Z1 \== Z2,
member(Z,Blanks),
Z \== Z1,Z \== Z2,
line(Z,Z1,X1),
line(Z,Z2,X2),!.
```

A fork is a pair of lines that have one empty square in common, one of the players tokens in each line and one other empty square in each line.

The code has a series of clauses which are partly keyed by the turn count, but this is not explicit:

```
choose_move(state(0, ..) ...) :- !.          % 1st move
choose_move(state(1, ..) ...) :- <cond>, !. % 2nd move case 1
choose_move(state(1, ..) ...) :- !.          % 2nd move default
choose_move(state(2, ..) ...) :- <cond>, !. % 3rd move case 1
choose_move(state(2, ..) ...) :- !.          % 3rd move default
choose_move(state(_, ..) ...) :- <cond>, !. % subsequent moves
choose_move(state(_, ..) ...) :- <cond>, !.
choose_move(state(_, ..) ...) :- <cond>, !.
choose_move(state(_, ..) ...) :- <cond>, !.
choose_move(state(_, ..) ...) :- ...         % default
```

It would be easier to read and easier to ensure that all the cases really have been taken care of by calling a predicate to determine the turn count and then hanging the remaining analysis from the results of that predicate.

```
choose_move(state(Turn, ..) ...) :- first_turn(Turn), !, ....
choose_move(state(Turn, ..) ...) :- second_turn(Turn), !, ....
choose_move(state(Turn, ..) ...) :- third_turn(Turn), !, ....
choose_move(state(Turn, ..) ...) :- later_turn(Turn), !, ....
```

P8 P8's code for choosing the computer's opening moves is also stylistically obscure, partly due to explicitly using integers for square positions. The following code takes either the centre square or a corner, but it does not mention them explicitly.

```
choose_move(State, o, Move-o) :-choose_move(State, o, Move-o) :-
```

```
        move_number( State, Number),
        ((Number == 1, Move = 1);
         (Number == 2,
          member(5-x, State),
          Move = 1);
         (Number == 2, Move = 5)),
        write('I move to position '),
        write(Move),nl,nl,
!.
```

P9 P9's code for choosing a move is obscured by not making explicit the order in which squares are being chosen. Here, P9 is making a default move. The code creates a list of empty squares, sorts them into best-first order and then chooses the first.

```
any_move(State,_,Move):-
        find_empty_sq(9,State,List),!,
        permute([5,3,1,7,9,2,8,4,6],List,List1),
        member(Move,List1).
```

The ordering implied by the list [5,3,1,7,9,2,8,4,6] is centre, corners and finally sides. (It is also not obvious, but **permute/3** does not generate permutations: instead it sorts the integers in **List** so they are in the order specified by the first argument.)

P8 P8 has used integers for characters in such a way as to make the program potentially awkward to port between Prologs. This could be seen as a failure to abstract.

```
display_board([P1, P2, P3, P4, P5, P6, P7, P8, P9], Status) :-
    name(Row1, [32, P1, 32, 124, 32, P2, 32, 124, 32, P3]),
    write(Row1),nl,
    board_throw_line,
    name(Row2, [32, P4, 32, 124, 32, P5, 32, 124, 32, P6]),
    write(Row2),
    display_status(Status),nl,
    board_throw_line,
    name(Row3, [32, P7, 32, 124, 32, P8, 32, 124, 32, P9]),
    write(Row3),nl,nl.
```

This code could have been written so that characters were declared in one place.

```
blank(32).
line(124).

display_board([P1, P2, P3, P4, P5, P6, P7, P8, P9], Status) :-
    blank(Blank), line(Line),
    name(Row1,
         [Blank, P1, Blank, Line,
          Blank, P2, Blank, Line,
          Blank , P3]),
    write(Row1),nl, ...
```

The code could have been written even more portably (though perhaps less efficiently) by using **name/2** to generate the characters.

```
blank(Blank) :- name(' ', [Blank]).
line(Line) :- name('|', Line).
```

P8 did use **name/2** when reading the user's input but not when generating output. That is, to translate the student's ASCII input to an ordinary atomic integer, P8 correctly used:

```
ascii_2_char(Ascii, Char) :- name(Char, [Ascii]).
```

but numbers were translated for ASCII output by addition, e.g.

```
get_display_char(P - n, Ascii) :- Ascii is P + 48.
```

instead of the portable:

```
get_display_char(P - n, Ascii) :- name(P, [Ascii]).
```

Perhaps this was a case of not seeing that **name/2** is reversible, or perhaps the student thought the code would be more efficient.

Effect: non-portable.

P2 Here the student is trying to make a concept, that of a **display square**, explicit. The display square is calculated from the square, but instead of using a named relation the student has used an 'or' branch and a named variable `Sq_disp`.

```
show_square(Sq) :-
  (var(Sq), Sq_disp =' ';
   Sq = o_us, Sq_disp = 'O';
   Sq = x_them, Sq_disp = 'X'),
   write(' '),
   write(Sq_disp),
   write(' ').
```

P2 has used variable names rather than relations to make a concept explicit.

P6 The failure to hide implementation details is the obverse of a failure to make concepts and relations explicit. What is explicit is the way in which concepts have been implemented, not the relations themselves. The code is written at too low a level. The effect is likely to make code unreadable. It is more difficult to reconstruct the purpose of the code and to verify that it does correspond to the user's intentions. It is also more difficult to change the underlying data representations while making only local changes to the code.

P6 wishes to identify a line in which the computer has taken a square and one other square in that line is free. Empty squares are identified by integers.

```
one_in_winset([o, N, _], N):- integer(N).
etc.
```

Here the built-in general-purpose predicate `integer/1` has been substituted for the more specific `empty_square/1` relation.

A.1.2 Scattering tasks through the code

P2 In some programs it is necessary to translate from the user's input to a useful square identifier for making a move. Once this is done there should be no

need to translate the identifier any further, i.e. the translation should be done in `choose_move/3` or `move/3` but not in both.

```
choose_move(State, x_them, m(Row/Col, x_them)) :-
  write('Choose your move (a-i)'),
  get(Input),
  Pos is Input - 97,
  Row is Pos div 3,
  Col is Pos mod 3.

move(m(Row/Col, Player), State, State) :-
  Rowarg is Row + 1,
  Colarg is Col + 1,
  arg(Rowarg, Rows, Line),
  arg(Colarg, Line, Player), !.
```

This is actually even more pointless than it looks because `choose_move/3` includes a test for the square's validity (not shown) which also has to increment the row and column identifiers.

This could also be interpreted as simply a bad choice of representation for the row and column identifiers: instead of using 0 to 2, they should be 1 to 3.

Effect: readability, verification, efficiency.

P3 Badly designed defaults also scatter functionality about. P3 explicitly writes code to choose strategies for responding to the opponent's first move. However, the clauses for this do not deal with the opponent taking the centre because that is dealt with by another clause that deals with the default move. This is difficult to read because the functionality is scattered about. It is also unsafe because if the student should alter the default behaviour it might no longer apply, or if the student inserts other strategies those might be applied incorrectly.

```
choose_move(State, Player, Move) :-
      second_move(State, Player, Move).
choose_move(State, Player, Move):-
      default_move(State, Player, Move).

% deal with every opening move except the centre
second_move(State, Player, Move) :- ..
```

Effect: readability, robustness.

A.1.3 Repetition of code

P9 P9 repeated an identical message in several clauses of `choose_move/3`.

```
choose_move(State,computer,Move/computer):-
        must_win(State,computer,Move),
        write('Computer has moved!'),nl,nl,!.
choose_move(State,computer,Move/computer):-
        prevent_win(State,computer,Move),
        write('Computer has moved!'),nl,nl,!.
choose_move(State,computer,Move/computer):-
        any_move(State,computer,Move),
        write('Computer has moved!'),nl,nl,!.

must_win(State,Player,Move):-
        find_empty_sq(9,State,List) ...
prevent_win(State,_,Move):-
        find_empty_sq(9,State,List) ...
any_move(State,_,Move):-
        find_empty_sq(9,State,List) ...
```

This would be more sensibly structured as:

```
choose_move(State,computer,Move/computer):-
        find_empty_sq(9,State,List),
        choose_move_sub(State,List,Player,Move),
        write('Computer has moved!'),nl,nl.

choose_move_sub(State,List,Player,Move) :-
        must_win(State,List, Player,Move).
choose_move_sub(State,List,Player,Move) :-
        prevent_win(State,List,_,Move).
choose_move_sub(State,List,Player,Move) :-
        any_move(State,List,_,Move).
```

P8 P8 has written almost identical clauses for detecting an incomplete line of its own and for detecting the user's incomplete line. Most programs used the same procedures for both of these, keyed by the relevant player's token. P8's

code refers explicitly to each token so that this code only works if the user plays x and the computer o.

```
block_move([], _) :- !, fail.
block_move([H | _], Move) :-
    block(H, Move),
    !.
block_move([_ | T], Move) :-
    block_move(T, Move).

block([P-n, _-x, _-x], P).
block([_-x, P-n, _-x], P).
block([_-x, _-x, P-n], P).

win_move([], _) :- !, fail.
win_move([H | _], Move) :-
    win(H, Move), !.
win_move([_ | T], Move) :-
    win_move(T, Move).

win([_-o, _-o, P-n], P).
win([_-o, P-n, _-o], P).
win([P-n, _-o, _-o], P).
```

P8 Two clauses, one for dealing with the computer's opening moves and one for dealing with subsequent moves, are not especially well structured. Code for producing a message to the user is repeated in both clauses. Each clause is cut to prevent any failure into the next one, although they deal with quite different situations.

```
choose_move(State, o, Move-o) :-
        move_number( State, Number),
        ((Number == 1, Move = 1);
         (Number == 2,
          member(5-x, State),
          Move = 1);
         (Number == 2, Move = 5)),
        write('I move to position '),
        write(Move),nl,nl,
!.
choose_move(State, o, Move-o) :-
    s_2_board(State, Board),
```

```
    (win_move(Board, Move);
     block_move(Board, Move);
     other_move(Board, Move);
     member(Move-n, State)),
    write('I move to position '),
    write(Move),nl,nl,
    !.
```

Effect: readability, verification.

P9 P9 has created procedures which do differ in structure and behaviour but whose function does not differ.

```
must_win(State,Player,Move):-
   find_empty_sq(9,State,List),
   permute([5,3,1,7,9,2,8,4,6],List,List1),!,
   member(Move,List1),
   empty_sq(Move,State),
   move(Move/Player,State,State2),
   win_state(State2,Player).

prevent_win(State,_,Move):-
   find_empty_sq(9,State,List),
   permute([5,4,6,2,8,3,1,7,9],List,List1),!,
   member(Move,List1),
   move(Move/user,State,State1),
   win_state(State1,user).
```

The calls to `permute/3` may return different orders, but since any square returned by these procedures is a square that must be taken to avoid losing, there is no reason to order the squares. The call to `empty_sq/2` in `must_win/3` also performs no function, since the list only contains empty squares.

A.1.4 Repetition of execution

P9 P9 derived a list of empty squares from a nested lines structure once for each procedure called from `choose_move/3`. This could have been computed just once and passed to the inner procedures.

Effect: efficiency.

A.1.5 Failing to validate an input argument

Validation of user input consists of type checking, range checking and checking that the selected square is empty. If any arithmetic analysis or manipulation is to be done on the inputs then they must be type checked explicitly, since the Prolog interpreter stops processing altogether if arithmetic is done on non-numeric values.

P9 P9 asks the user to type in an integer and then performs a range check on the integer without first checking that it is indeed an integer. Prolog will produce a strange error message if the user types in something that is not an integer.

```
ask_user_move(Move):-
        write('Enter your move here (1-9) ->'),
        read(Move),
        Move > 0, Move < 10.
```

P3 P3 asks the user to type in a pair of integers and then performs a range check on the integers without first checking that they are indeed integers. Prolog will produce a strange error message if the user types in a pair of items that are not integers.

```
check_move([1,1],Player,[line(Player,_,_)|_]).
check_move([2,1],Player,[line(_,Player,_)|_]).
check_move([3,1],Player,[line(_,_,Player)|_]).
check_move([X,Y],Player,[_|Tail]) :-
        Y > 1,
        Z is Y - 1,
        check_move([X,Z],Player,Tail).
```

P3 P3 has code for validating the user's move in which inputs can be out of range.

```
check_move([1,1],Player,[line(Player,_,_)|_]).
check_move([2,1],Player,[line(_,Player,_)|_]).
check_move([3,1],Player,[line(_,_,Player)|_]).
```

```
check_move([X,Y],Player,[_|Tail]) :-
        Y > 1,
        Z is Y - 1,
        check_move([X,Z],Player,Tail).
```

This code is actually very unsafe. The first argument has been input by the user, and this code is intended to validate that input. It is only intended to work on rows and columns between 1 and 3, but it can be fed larger values for the second co-ordinate in which case the token will be placed in some other line. This could be considered a data range error. The values should range over the number of rows and the number of columns, but in fact they range over the length of a line and the number of lines.

A.1.6 Code that is never called

P4 A single Prolog procedure has a given name and a fixed number of arguments. Two different procedures may have the same name if they have different numbers of arguments. If a student accidentally adds an extra argument to some clauses of a procedure, leaving the others the same and using the same call, then the clauses with the extra argument will never be used (i.e. they are treated as a separate procedure that just happens not to be called).

P4's implementation of `choose_move/3` includes some clauses that have an extra argument and are therefore never called.

A.1.7 Mis–use of data constructors

P2 P2 has tried to use the same code in two different contexts. Code which generates (row,column) co-ordinates is also used to index into a nested lines structure. The row co-ordinate is re–used to determine whether rows, columns or diagonals are to be used. The column co-ordinate is re–used to index to the relevant line. This is a mis–use, especially since the (row,column) pairs generate a non–existent line, the third diagonal.

Here is the code that represents the co-ordinates:

```
co_ord(X/Y) :- half_coord(X), half_coord(Y).
half_coord(1).
half_coord(2).
half_coord(3).
```

(The numbering has been altered for brevity. This does not affect the meaning of the code, which incremented all the values before using them).

Here is some code which uses them as (row,column) co-ordinates to find out what is in a particular *square* in the board:

```
end_state(State, draw) :-
    \+ (co_ord(Some_coord),
        occupies(State, Some_coord, Occ),
        var(Occ)).
occupies(board(Rows,_,_), Row/Col, Entry) :-
    arg(Row, Rows, Line),
    arg(Col, Line, Entry).
```

Here the co-ordinates are used to index in to the Rows only. By contrast, the other clause for `end_state/2` uses the same co-ordinates to identify each *line*:

```
end_state(State, win(Winner)) :-
    co_ord(GroupId/LineId),
    arg(Group, State, RowsOrColsOrDiags),
    arg(LineId, RowsOrColsOrDiags, line(X, Y, Z)),
    X == Y, Y == Z,
    Winner = X.
```

This latter use is incorrect. The indices are applied to the whole structure, not just the rows. Given the student's board structure:

```
    board(rows(L1, L2, L3),
          cols(L4, L5, L6),
          diags(L7, L8))
```

the co-ordinate (3,3) retrieves no line at all. This code depends on the built-in `arg/3` predicate handling indices that are out of range as a failure, not as an error.

Effect: readability, verifiability, robustness across Prologs.

P1

```
move(+Move, +State, -NewState)

move([Place|Player], State, NewState):-
  ....
```

Here the move looks remarkably like a list of some indefinite length in which `Place` is an element and `Player` is the tail of the list. However in fact the move is a pair, whose elements are `Place` and `Player`. This is a confusing misuse of Prolog's data constructors but it is not forbidden.

Effect: readability, verifiability, robustness across Prologs.

P3 P3 asks the user to type in a pair of integers and then performs range checks on the integers without first checking that they are indeed integers. P3 also uses an inappropriate range check for one of the values. These errors are classified as a failure to validate an input argument and so they are described in Section , but they can also be seen as a mis-use of data structures.

Effect: program will crash on some inputs.

A.2 Design in an unfamiliar problem domain

P1 P1's attempt to implement minimax is subverted by a poor choice of evaluation criterion.

Effect: success of player.

P1 Attempt to use minimax from the first move. This runs extremely slowly.

Effect: speed.

P6 P6 wishes to identify a line in which the computer has taken a square and one other square in that line is free. It is arguable that this is a bad strategy because it does not require that both the other squares are empty. If the opponent holds the third square then the move is (probably) useless. to take a square in a line which contains none of its opponent's tokens. The code could be modified easily to take a square in a line which contains none of its opponent's tokens:

```
one_in_winset([o, N, X], N):- integer(N), \+ opponent(X).
one_in_winset([N, o, X], N):- integer(N), \+ opponent(X).
one_in_winset([X, N, o], N):- integer(N), \+ opponent(X).
```

This would suffice as a heuristic, although the case analysis would still be incomplete for a look-ahead which needed to look at all the possible placements.

Effect: success of player.

P4 P4 has written code which picks out a three different items from a list. The code uses a generate-and-test algorithm instead of a subtraction algorithm (i.e. subtract an item from the list, then subtract the next from the remainder of the list, and so on).

```
choose_move(state(_,_,_,Blanks,_),Player, move(Player,Z)):-
   ...,
   member(Z1,Blanks),
   member(Z2,Blanks),
   Z1 \== Z2,
   member(Z,Blanks),
   Z \== Z1,Z \== Z2,
   ...
```

Effect: speed.

A.2.1 Failure to realise all constraints

P6 P6 wishes to identify a line in which the computer has taken a square and one other square in that line is free. o is the computer's token, and empty squares are identified by integers. The clauses are as follows:

```
one_in_winset([o, N, _], N):- integer(N).
one_in_winset([N, o, _], N):- integer(N).
one_in_winset([_, N, o], N):- integer(N).
```

Various cases have been omitted, for example:

```
one_in_winset([N, _, o], N):- integer(N).
```

This may well be a result of lack of abstraction — the student has tried to list all the cases explicitly and has forgotten some of them. The explicit relations, i.e. a single square occupied by the computer's token plus a single empty square, are not explicit.

Effect: success of player.

P8 P8 has used very similar code to P6, although it is more likely that P8's code does reflect the programmers intentions. P8's code identifies a line in which the computer has taken a square and both other squares in that line are free. `o` is the computer's token, and empty squares are identified by the special token `n`. The squares also contain a square identifier. The clauses are as follows:

```
o_move(+Line, -Place)

o_move([_-o, _-n, P-n], P).
o_move([P-n, _-n, _-o], P).
o_move([P-n, _-o, _-n], P).
```

The code deals with all the possibilities but it is still somewhat heavily compiled. We don't know for sure if there is a reason for choosing squares in this particular order.

Effect: unreadable, success of player.

A.2.2 Failure to realise interactions

P6 P6's code attempts to prevent a win by the user before completing a line of its own.

Effect: success of player.

A.3 Design in an unfamiliar language

A.3.1 Misconceptions about Prolog primitives

Analogy

P7 P7 seemed very keen to avoid using Prolog's pattern matching in the heads of clauses, to perform case analysis by multiple clauses, or even to allow a Prolog procedure to return a value without using an explicit =. This led to code such as the following for putting a token into the board:

```
move(Move, Player, State, State1):-
        Player = user, !, update(Move, x, State, State1);
                          update(Move, o, State, State1).

update([R,C], X, [R1, R2, R3], State1):-
        R = a, !, up(C, X, R1, Nrow), State1 = [Nrow, R2, R3];
        R = b, !, up(C, X, R2, Nrow), State1 = [R1, Nrow, R3];
        up(C, X, R3, Nrow), State1 = [R1, R2, Nrow].

up(C, X, [C1, C2, C3], Nrow):-
        C = 1, !, Nrow = [X, C2, C3];
        C = 2, !, Nrow = [C1, X, C3];
        Nrow = [C1, C2, X].
```

This could have been written correctly in Prolog, without need for cuts:

```
move(Move, user,     State, State1):-
        update(Move, x, State, State1).
move(Move, computer, State, State1):-
```

```
        update(Move, o, State, State1).

update([a,C], X, [R1, R2, R3], [Nrow, R2, R3]):-
        up(C, X, R1, Nrow).
update([b,C], X, [R1, R2, R3], [R1, Nrow, R3]):-
        up(C, X, R2, Nrow).
update([c,C], X, [R1, R2, R3], [R1, R2, Nrow]):-
        up(C, X, R3, Nrow).

up(1, X, [C1, C2, C3], [X, C2, C3]).
up(2, X, [C1, C2, C3], [C1, X, C3]).
up(3, X, [C1, C2, C3], [C2, C2, X]).
```

This code would have been improved still further by being recursive, although the letters a to c would have to be replaced by integers 1 to 3 to really make a recursion work.

A.3.2 Misconceptions about Prolog techniques

List recursions

P1 P1's attempt to find the maximum value in a list of integers is a good example of a bad implementation of a list recursive technique. This code requires an extra input which must be supplied by the caller and must be smaller than any integer in the list. This code is a variant of a technique for list recursion that works well for computing e.g. the sum of a list of numbers but cannot be used to compute the maximum or minimum (O'Keefe, 1990). Instead of using the correct technique, the user has patched the incorrect technique, resulting in an idiosyncratic implementation that is fragile.

The student's version:

```
find_max(X,[],X).
find_max(MaxSoFar,[H|T],Result):-
     H > MaxSoFar,
     find_max(H,T,Result).
find_max(MaxSoFar,[_|T],Result):-
     find_max(MaxSoFar,T,Result).
```

```
Call find_max(Max, [3,5,1,3,2], 9999)
```

A correct version:

```
find_max(Max, [H|List]) :-
find_max(H, List, Max).

find_max(X,[],X).
find_max(MaxSoFar,[H|T],Result):-
     H > MaxSoFar,
     find_max(H,T,Result).
find_max(MaxSoFar,[_|T],Result):-
     find_max(MaxSoFar,T,Result).

Call find_max(Max, [3,5,1,3,2])
```

In the terms used in (O'Keefe, 1990), the student has taken a technique which is suitable for functions which have a left identity and has applied the technique directly to a function which has no left identity by supplying a context-dependent left identity. Richard's book describes a more appropriate technique for such problems.

Effects: dependent on caller, unreadable.

P5 P5's code to find the n'th element of a list depends on a misunderstanding of the Prolog technique for recursively accumulating in the head of the clause. This program should be an application of a standard list recursive accumulator technique which accumulates the items implicitly but efficiently. The student explicitly implemented an accumulator for the front of the list.

The student's implementation:

```
rep_nth(+Token, +Position, +List, +[], -List)

rep_nth(Item,1,[_|T],Sofar,Answer)
        append(Sofar,[Item|T],Answer).
rep_nth(Item,N,[X|L],Sofar,Answer) :- R is N - 1,
        append(Sofar,[X],New),
        rep_nth(Item,R,L,New,Answer).
```

The recursive call explicitly accumulates the items before the item to be replaced. The base appends the replaced item plus the items after it to the items before it. The repeated use of **append/3** to add a single item to the end of the list is suspect and inefficient.

The correct technique accumulates the items implicitly but efficiently using a standard accumulator technique:

```
rep_nth(+Token, +Position, +List, -List)

rep_nth(Item,1,[_|T],[Item|T]).
rep_nth(Item,N,[X|L],[X|Answer]) :-
        N > 1,
        R is N - 1,
        rep_nth(Item,R,L,Answer).
```

The procedure **rep_nth/5** needs an extra input which is an empty list. This is unsafe and should be hidden.

Effect: efficiency, dependent on caller.

P7 This code attempts to collect a list of items from different places. It collects the positions of a particular token (**Mark**) from a board that consists of a list of rows. The result is to be a single list of positions. The student has built three lists separately and then joined them together using **append/3** instead of accumulating the results into a single list.

```
getx(Mark, [R1, R2, R3], Pos_list):-
        belong(Mark, R1, 1, Row1),
        belong(Mark, R2, 4, Row2),
        belong(Mark, R3, 7, Row3),
        ap(Row1, Row2, Row3, Pos_list).

belong(_, [], _, []):- !.
belong(H, [H|T], Pos, [Pos|R]):- !,
        Pos1 is Pos + 1,
belong(X, [_|T], Pos, R):-
        Pos1 is Pos + 1,
        belong(X, T, Pos1, R).
```

```
ap(Xs, Ys, Zs, State):-
        append(Xs, Ys, Inter),
        append(Inter, Zs, State).
```

Effect: efficiency.

P6 P6's code for finding the next empty square has a structural bug in which a procedure that should be recursive is not.

```
computer_turn(State, Position):- avail(State, Position).

avail([Line|_], Position):- avail_winset(Line, Position).
avail([_|Lines], Position):- avail_winset(Lines, Position).

avail_winset([Position|_], Position):- integer(Position), !.
avail_winset([_|Rest], Position):- avail_winset(Rest, Position).
```

The second clause of `avail/2` should be recursive. This is really a type error: `avail/2` is intended to be called on the whole board whereas `avail_winset/2` should only be called on individual lines. In theory, this code would fail if the first row is entirely full. In practice this code is never called with the first row full: all the possibilities are trapped by earlier case analysis.

Counters and reversibility

P1 Here we have an example of a poorly chosen reversible technique.

P1 wishes to use to the same code to find the n'th square in the board and to find the position of particular squares.

```
pick_nth(+Position, +List, -Token)
pick_nth(-Position, -List, +Token)

pick_nth(1,[X|_],X).
pick_nth(N,[_|T],X):-
        pick_nth(N1,T,X),
        N is N1 + 1.
```

This technique works with both sets of modes but it is inefficient, since it cannot use Prolog's tail recursion optimisation.

A simple rewrite into a tail recursive form will not work to find the position of an element.

```
pick_nth(+Position, +List, -Token)

pick_nth(1,[X|_],X).
pick_nth(N,[_|T],X):-
        N is N1 + 1,
        pick_nth(N1,T,X).
```

A tail recursive version that is reversible would require an extra argument and a sub–procedure.

```
pick_nth(+Position, +List, -Token)
pick_nth(-Position, -List, +Token)

pick_nth(Position, List, Token) :-
pick_nth(Position, 1, List, Token).

pick_nth(N, N, [X|_], X).
pick_nth(N, NPrev, [_|T], X) :-
N < NPrev,
Next is NPrev+1,
pick_nth(N, NPrev, T, X).
```

Case analyses

P1 The student has attempted to write just two clauses to announce the result of the game. The first clause deals with a draw, the second with either player winning.

```
announce(draw):-
        write('The game is a draw'), nl.
announce(Winner):-
        write(Winner),
        write(' wins!'), nl.
```

This code is unsafe because given a draw it can fail back into the second clause. There are several possible solutions. The first possibility is to cut the first clause. This could be termed a 'case analysis with cuts' technique (see below). This would be something of a bodge. The second solution is to use the form of the Result variable to distinguish between the cases, e.g. the Result could be either `draw` or `win(Winner)`. This is the solution that O'Keefe proposes. A third solution would be to have three separate clauses, one for each possible result. This would be less concise and could lead to a less robust solution if code is repeated.

Effect: unreliable on failure.

P5 One common form of case analysis with defaults is when recursing on a number. It is easy to forget to prevent subsequent failure from causing a later clause to be called.

```
rep_nth(+Token, +Position, +List, +[], -List)

rep_nth(Item,1,[_|T],Sofar,Answer) :-
        append(Sofar,[Item|T],Answer).
rep_nth(Item,N,[X|L],Sofar,Answer) :- R is N - 1,
        append(Sofar,[X],New),
        rep_nth(Item,R,L,New,Answer).
```

Among other errors, the base case of this code can fail through to the second clause. In fact, this code does not give spurious answers or loop forever on failure or on being given a negative argument, although on first sight it should. This is because the third argument shrinks on each recursive call. Eventually it reaches nil, no clause matches and the procedure fails.

Effect: no obvious effect on behaviour.

P1 This code should give only one solution. It does not because it does not guard the third clause properly.

```
find_max(+LargeNegativeInteger, +IntegerList,-MaxInteger)

find_max(X,[],X).
find_max(MaxSoFar,[H|T],Result):-
     H > MaxSoFar,
     find_max(H,T,Result).
find_max(MaxSoFar,[_|T],Result):-
     find_max(MaxSoFar,T,Result).
```

On failure it is likely to give extra spurious results, as a result of using a "defaulting" technique between the second and third clauses.

Effect: spurious results on failure.

P2

```
show_square(Sq) :-
  (var(Sq), Sq_disp =' ';
   Sq = o_us, Sq_disp = 'O';
   Sq = x_them, Sq_disp = 'X'),
   write(' '),
   write(Sq_disp),
   write(' ').
```

This code is unsafe on failure, as the student has used an uninstantiated variable for the empty square but has failed to realise that `Sq = o_us` will succeed if `Sq` is uninstantiated.

Effect: spurious results on failure.

P6 The code puts a token into the appropriate square in a line. The first clause deals with the case in which the square is the correct one, the second if not. There is no test in the second clause, so the cut in the first clause is red.

```
update_winset(Position, Entry, [Position|Rest], [Entry|Rest]):- !.
update_winset(Position, Entry, [N|Rest], [N|NewRest]):-
        update_winset(Position, Entry, Rest, NewRest).
update_winset(_, _, [], []).
```

P2 Here the aim is to announce the result of the game.

P2's code has a final default case:

```
announce(win(x_them)) :- write('YOU WIN'), !.
announce(win(o_us)) :- write('I WIN'), !.
announce(_) :- write('WE DRAW').
```

Firstly, there are only three legitimate values for the result. If the third value, "draw", had been used explicitly in the head of the third clause, there would have been no need to cut the other two clauses. Secondly, if the student had felt it necessary to cut them anyway, the cuts should have come before the `write/1` statements so as to indicate that their role was case analysis.

P3 Here the aim is to announce the result of the game.

P3 has a badly placed cut:

```
announce(Player) :- var(Player),write('The game was a draw'),nl,!.
announce(Player) :- write(Player),write(' wins'),nl.
```

The second clause does not explicitly deal with a Player that is instantiated. Instead the first clause is cut. The cut should come after the first call.

P8 P8 has two clauses, one for dealing with the computer's opening moves and one for dealing with subsequent moves. The second clause does not explicitly deal with later moves. Instead the first clause is cut.

```
choose_move(State, o, Move-o) :-
        move_number( State, Number),
        ((Number == 1, Move = 1);
         (Number == 2,
          member(5-x, State),
          Move = 1);
         (Number == 2, Move = 5)),
        write('I move to position '),
        write(Move),nl,nl,
!.
```

```
choose_move(State, o, Move-o) :-
    s_2_board(State, Board),
    (win_move(Board, Move);
     block_move(Board, Move);
     other_move(Board, Move);
     member(Move-n, State)),
    write('I move to position '),
    write(Move),nl,nl,
    !.
```

Loops to validate inputs

Consider the code for asking a question to which the valid answers are either "yes" or "no". Two standard techniques for this are use of a failure-driven loop and recursion on failure.

Most noughts and crosses programs ask the user if s/he wishes to play first, and return the name of the player whose turn is first.

P9 Here is a good example of using a failure-driven loop for this:

```
go_first(Player) :-
    repeat,
        write('Do you want to go first'),
        read(FirstP),
        translate_player(FirstP, Player).
translate_player(yes,human).
translate_player(no, computer).
```

The `repeat` ensures that if anything other than "yes" or "no" is offered, the question is repeated.

Here are two examples of unsafe code which may behave badly if the user types in neither "yes" nor "no".

P1 This example assumes a default value when the user types any answer except "yes":

```
gofirst(human) :- write('Do you want to go first'),
                  read(yes).
gofirst(computer).
```

P8 This example fails (and indeed causes the whole program to fail) if the user types in anything except "yes" or "no":

```
go_first(Player) :-
        write('Do you want to go first'),
        read(FirstP),
        translate_player(FirstP, Player).
translate_player(yes, human).
translate_player(no, computer).
```

Both of these examples may be considered as bad choices of technique. It is debatable about which mistake is "worse" – in this case defaulting is probably slightly better behaviour but in other contexts outright failure may be better than continuing with an invalid input.

A.3.3 Choice of data structure

P7 P7 chose to represent the board by a data structure in which it is difficult to detect lines, i.e. the list of rows. This data structure could have been translated into some easier one. Alternatively the 8 options could have been listed exhaustively. Instead P7 chose to translate to another awkward formalism which had to be processed inefficiently. P7 obtained a list of the positions held by a player, then permuted the list against a set of possible line positions.

P7 This code is a good example of how suitable initial choices can become subverted for no very apparent reason. P7 has used a list of rows for the board, in which each row is also a list and empty squares are represented by the atom 'e'. The program has asked the user to supply (row, column) indices for the move. This code validates the user's input, i.e. that the indices must be correct (a to c for rows, 1 to 3 for columns) and they must represent an empty square.

```
valid(State, [R,C]):-
        getx(e, State, Xpos),
        row(R, N),
        column(C, M),
```

```
        P is N*3+M,
        member(P, Xpos).

row(a, 0).      row(b, 1).      row(c, 2).
column(1, 1).   column(2, 2).   column(3, 3).

getx(Mark, [R1, R2, R3], Pos_list):-
        belong(Mark, R1, 1, Row1),
        belong(Mark, R2, 4, Row2),
        belong(Mark, R3, 7, Row3),
        ap(Row1, Row2, Row3, Pos_list).

belong(_, [], _, []):- !.
belong(H, [H|T], Pos, [Pos|R]):- !,
        Pos1 is Pos + 1,
belong(X, [_|T], Pos, R):-
        Pos1 is Pos + 1,
        belong(X, T, Pos1, R).

ap(Xs, Ys, Zs, State):-
        append(Xs, Ys, Inter),
        append(Inter, Zs, State).
```

This code contains several errors of programming style. It is not clear why the row index is a letter. If we accept that, we would expect the code for `valid/2` to translate the co-ordinates so that they are both numeric and then use the co-ordinates to pick out recursively first the row and then the square in the row. In the larger context of the program, we would also expect `valid/2` to return the translated indices for future use in putting the token into place.

Instead of this, the code takes a long detour. First of all it builds a list of all the positions of the empty squares in the board as numbers from 1 to 9. Then it translates the indices as typed in first into numbers and then into a single number between 1 and 9. Finally it looks for this number in the list of empty square positions. Having done all this, it still does not return either of the translated positions for future reference.

It appears that the student has failed to realise that the original data representations were appropriate for the task and for each other, and has spent a lot of effort in translating them instead. The user also failed to realise that original

representation is not ideal for placing a move in the board, although an intermediate representation (integers for the row and column) is. Therefore the program also translates the computer's move into a (letter, number) pair before making the move.

P3 The use of an uninstantiated variable with a meaning leads to the need for meta-predicates and extra control.

```
end_state(State, Winner) :- .. find the Winner ..
end_state(State, _) :- .. recognise a draw ..

announce(Winner) :- var(Winner), write(draw), !.
announce(Winner) :- write(Winner), write(' wins').
```

Here no explicit value is given to the end of game variable if the game was a draw. As a result the student must use meta-logic and cuts in the predicate that analyses that variable to announce the winner. The code for `end_state/2` could easily have returned the atom "draw" explicitly.

Later on in the same program, we have similar problems:

```
win_pattern(line(Blank,A,B),Player,Player1) :-
    nonvar(A),
    nonvar(B),
    A = B,
    A = Player1,
    var(Blank),
    Blank = Player,!.
```

This causes a lot of verbiage in the program because simple pattern matching cannot be used to look for patterns in the board. This is exacerbated by the student not using `==`.

In order to recognise patterns that include empty squares P3 must use extra-logical predicates such as `var/1`, `nonvar/1`, `==` and `=`. The student has not tried to hide the underlying representation by writing predicates whose role which infer whether a square is occupied or empty.

```
opening_pattern(line(A,B,C),
                line(D, E, F), G, Player, Opponent) :-
    nonvar(A),A = Opponent,
    var(B),var(C),
    blank_line(line(D,E,F)),
    blank_line(G),
    E = Player.
```

The program is peppered with calls to `var/1` and `nonvar/1` in order to distinguish full and empty squares.

The program also used an uninstantiated variable for a drawn result.

P9 P9 also used an uninstantiated variable to represent an empty square, but kept the code hidden inside procedures which tested that the square at a particular position is empty, which is an improvement.

P2 P2 wishes to use a row, column notation in which rows and columns are numbered from 0 to 2. However, Prolog's indexing actually requires numbers from 1 to 3 and there is no part of the code that actually requires the use of numbers 0 to 2 instead. The result is a lot of unnecessary incrementing.

Appendix B

Demonstration Program 1

This is the example program described in Chapters 4 and 5 and in Section 6.3. This example has been created for demonstration purposes and has been simplified.

It consists of the top level predicates plus three of the main predicates that get called — those of choosing a move, making the move and swapping players. It leaves out the other major tasks such as initialising the board, displaying the board, obtaining the user's move and dealing with the end of the game. The move choosing is as simple as is consistent with the specification. The program will complete a line of two or block an opponent's line of two, but it will not use any other heuristics and will otherwise just pick any empty square.

```
/*================================================================
Here we have an example of something that we would like to be able
to match.

It consists of the top level predicates plus three of the main
predicates that get called.  These predicates are
        choose_move/3
        move/3
        next_player/2
next_player/2 is only included because it's needed as part of
choose_move/3.

For simplicity, at present we leave out two important bits of choosing
a move. First of all, we leave out getting the user's move. Secondly,
we leave out any extra heuristics for choosing a move: the only ones
we have are the ones in the specification, i.e. complete a line, block
```

an opponent's line, and then grab any empty square. We could add either of these, either as extra bits to recognise or as extra unrecognisable "soup".

This implementation uses the following data representations:

The State (i.e. the board) is a list of squares. Each square is an atom, either a player token (o or x) or an empty square (empty).

The Player is simply represented by an atomic token, o or x. (This implies that the user would not have been allowed to choose which token to play.)

The Place to which a token is moved is represented by an integer, the position of the square in the board.

It is awkward to identify potential lines using that representation for the board. An intermediate representation has been used to identify potential lines, in which a line consists of three squares and each square consists of the token and its integer position in the board.

Other decisions about these data structures would use different predicates.

Predicates: 15 Clauses: 27

```
Call tree:
  1 play/1
  2    initialise/2  % UNDEFINED
  3    display_game/2  % UNDEFINED
  4    play/3
  5       end_state/2  % UNDEFINED
  6       announce/1  % UNDEFINED
  7       choose_move/3
  8          i_win/3
  9             fill_a_pair/3
 10                find_line/2
 11                pair_and_blank/3
 12                   empty_square/1
 13          block_lose/3
 14             next_player/2
 15             fill_a_pair/3  % see 9
 16          next_empty_square/2
 17             next_empty_square/3
 18                empty_square/1  % see 12
 19                next_empty_square/3  % see 17
 20       move/3
 21          move/4
 22             move/4
 23       display_game/2  % UNDEFINED
```

```
 24       next_player/2  % see 14
 25       play/3  % see 4

*****************************************************************/

play(Result):-                  % To play, with some final Result:
    initialise(State, Player),   % First make an initial game state
    display_game(State, Player), % Then display the game
    play(State, Player, Result). % Then play the game, with
                                 % some Result

play(State, _, Result):-
    end_state(State, Result),  % If some terminating state
    !,                         % is reached
    announce(Result).          % Then just report the result
play(State, Player, Result):-  % Otherwise, the current player
    choose_move(State, Player, Move),  % chooses a move.
    move(Move, State, State1), % That move is implemented
    display_game(State1, Player),   % The game is redisplayed
    next_player(Player, Player1),   % We find out who the next
    !,                              % player is
    play(State1, Player1, Result).  % And give him/her/it a turn

/*****************************************************************
  choose_move(+State, +Player, -move(Place, Token))
  choose_move(list(atom), atom, move(integer, atom))

  Move a player's Token to the Place'th square in State,
  giving State1
*****************************************************************/
choose_move(State, Player, move(Place, Player)) :-
    i_win(State, Player, Place),
    block_lose(State, Player, Place),
    next_empty_square(State, Place).

/*****************************************************************
  i_win(+State, +Player, -Blank)
  i_win(list(atom), atom, integer)

  Look for a line in the board which contains two of my
  tokens plus an empty square. If there is one, then
  return the position of the empty square (to take it).
*****************************************************************/
i_win(State, Player, Place) :- fill_a_pair(State, Player, Place).

/*****************************************************************
  block_lose(+State, +Player, -Blank)
  block_lose(list(atom), atom, integer)

  Look for a line in the board which contains two of my
  opponent's tokens plus an empty square. If there is one, then
```

```
  return the position of the empty square (to take it).
*****************************************************************/
block_lose(State, Player, Place) :-
    next_player(Player, Opponent),
    fill_a_pair(State, Opponent, Place).

/*****************************************************************
  next_player(+Player, -Opponent)
  next_player(atom, atom)

  The other player.
*****************************************************************/
next_player(o, x).
next_player(x, o).

/*****************************************************************
  fill_a_pair(+State, +Player, -Blank)
  fill_a_pair(list(atom), atom, integer)

  Look for a line in the board which contains two of the Player's
  tokens plus an empty square. If there is one, then
  return the position of the empty square.
*****************************************************************/
fill_a_pair(State, Player, Blank) :-
    find_line(State, Line),              % generate a line
    pair_and_blank(Line, Player, Blank). % is there a pair?

/*****************************************************************
  find_line(+State, -Line)
  find_line(list(atom),
            line(pair(atom,integer),
                 pair(atom,integer), pair(atom,integer)))

  Obtain each line in the board, in turn. Each line consists of
  three squares, and each square consists of the contents of the
  square (i.e. player token or empty) plus its numeric position
  in the board.
*****************************************************************/
find_line([S1, S2, S3, S4, S5, S6, S7, S8, S9],
      line(sq(S1,1), sq(S2,2), sq(S3,3))).
find_line([S1, S2, S3, S4, S5, S6, S7, S8, S9],
      line(sq(S4,4), sq(S5,5), sq(S6,6))).
find_line([S1, S2, S3, S4, S5, S6, S7, S8, S9],
      line(sq(S7,7), sq(S8,8), sq(S9,9))).
find_line([S1, S2, S3, S4, S5, S6, S7, S8, S9],
      line(sq(S1,1), sq(S4,4), sq(S7,7))).
find_line([S1, S2, S3, S4, S5, S6, S7, S8, S9],
      line(sq(S2,2), sq(S5,5), sq(S8,8))).
find_line([S1, S2, S3, S4, S5, S6, S7, S8, S9],
      line(sq(S3,3), sq(S6,6), sq(S9,9))).
find_line([S1, S2, S3, S4, S5, S6, S7, S8, S9],
```

```
      line(sq(S1,1), sq(S5,5), sq(S9,9))).
find_line([S1, S2, S3, S4, S5, S6, S7, S8, S9],
      line(sq(S3,3), sq(S5,5), sq(S7,7))).

/*=============================================================
  pair_and_blank(+Line, +Player, -Place)
  pair_and_blank(line(atom-integer, atom-integer, atom-integer),
        atom, integer)

  Test whether line contains two tokens belonging to Player
  and an empty square. Return the position of the blank.
=============================================================*/
pair_and_blank(line(sq(Player,_), sq(Player,_), sq(Empty,Blank)),
          Player, Blank) :-
    empty_square(Empty).
pair_and_blank(line(sq(Player,_), sq(Empty,Blank), sq(Player,_)),
          Player, Blank) :-
    empty_square(Empty).
pair_and_blank(line(sq(Empty,Blank), sq(Player,_), sq(Player,_)),
          Player, Blank) :-
    empty_square(Empty).

/*=============================================================
  empty_square(+Square)
  empty_square(atom)

  Test if a square is empty
=============================================================*/
empty_square(empty).

/*=============================================================
  next_empty_square(+State, -Place)
  move(list(atom), integer)

  Find the position of the next empty square
=============================================================*/

next_empty_square(State, Place) :-
    next_empty_square(State, 1, Place).

next_empty_square([Empty|_Squares], Place, Place) :-
    empty_square(Empty).
next_empty_square([_Sq|Squares], SoFar, Place) :-
    Next is SoFar+1,
    next_empty_square(Squares, Next, Place).

/*=============================================================
  move(+move(Place, Token), +State, -State1)
  move(move(integer, atom), list(atom), list(atom))

  Move a player's Token to the Place'th square in State,
```

```
  giving State1
*****************************************************************/
move(move(Place, Token), State, State1) :-
    move(Place, Token, State, State1).

move(1, Token, [_Square|Squares], [Token|Squares]).
move(Place, Token, [Square|Squares], [Square|NewSquares]) :-
    Place > 1,
    P1 is Place-1,
    move(P1, Token, Squares, NewSquares).
```

Appendix C

Demonstration Program 2

This is the example program described in Sections 5.3 and 6.4. Its purpose is to demonstrate the use of clausal split to identify unfamiliar code and localise a bug.

The predicates for **next_empty_square** are different from those used in Appendix B to illustrate the use of clausal split, but it is otherwise identical to that code.

This example has been created for demonstration purposes and has been simplified.

```
/*================================================================
VARIANT: we use an unusual version of next_empty_square.

It consists of the top level procedures plus three of the main
procedures that get called.  These procedures are
        choose_move/3
        move/3
        next_player/2
next_player/2 is only included because it's needed as part of
choose_move/3.

This implementation uses the following data representations:

The State (i.e. the board) is a list of squares. Each square is an
atom, either a player token (o or x) or an empty square (empty).

The Player is simply represented by an atomic token, o or x. (This
implies that the user would not have been allowed to choose which
token to play.)
```

The Place to which a token is moved is represented by an integer, the position of the square in the board.

It is awkward to identify potential lines using that representation for the board. An intermediate representation has been used to identify potential lines, in which a line consists of three squares and each square consists of the token and its integer position in the board.

Other decisions about these data structures would use different procedures.

In this example everything has been kept as simple and as explicit as possible.

Procedures: 15 Clauses: 27

```
Call tree:
  1 play/1
  2    initialise/2  % UNDEFINED
  3    display_game/2  % UNDEFINED
  4    play/3
  5       end_state/2  % UNDEFINED
  6       announce/1  % UNDEFINED
  7       choose_move/3
  8          i_win/3
  9             fill_a_pair/3
 10                find_line/2
 11                pair_and_blank/3
 12                   empty_square/1
 13          block_lose/3
 14             next_player/2
 15             fill_a_pair/3  % see 9
 16          next_empty_square/2
 17             next_empty_square/3
 18                empty_square/1  % see 12
 19                next_empty_square/3  % see 17
 20       move/3
 21          move/4
 22             move/4
 23       display_game/2  % UNDEFINED
 24       next_player/2  % see 14
 25       play/3  % see 4

*****************************************************************/

play(Result):-                  % To play, with some final Result:
    initialise(State, Player),    % First make an initial game state
    display_game(State, Player), % Then display the game
    play(State, Player, Result). % Then play the game, with
```

```
                                % some Result
play(State, _, Result):-
    end_state(State, Result),  % If some terminating state
    !,                         % is reached
    announce(Result).          % Then just report the result
play(State, Player, Result):-  % Otherwise, the current player
    choose_move(State, Player, Move),   % chooses a move.
    move(Move, State, State1), % That move is implemented
    display_game(State1, Player),   % The game is redisplayed
    next_player(Player, Player1),   % We find out who the next
    !,                              % player is
    play(State1, Player1, Result).  % And give him/her/it a turn

/*=============================================================
  choose_move(+State, +Player, -move(Place, Token))
  choose_move(list(atom), atom, move(integer, atom))

  Move a player's Token to the Place'th square in State,
  giving State1
=============================================================*/
choose_move(State, Player, move(Place, Player)) :-
    i_win(State, Player, Place),
    block_lose(State, Player, Place),
    next_empty_square(State, Place).

/*=============================================================
  i_win(+State, +Player, -Blank)
  i_win(list(atom), atom, integer)

  Look for a line in the board which contains two of my
  tokens plus an empty square. If there is one, then
  return the position of the empty square (to take it).
=============================================================*/
i_win(State, Player, Place) :- fill_a_pair(State, Player, Place).

/*=============================================================
  block_lose(+State, +Player, -Blank)
  block_lose(list(atom), atom, integer)

  Look for a line in the board which contains two of my opponent's
  tokens plus an empty square. If there is one, then
  return the position of the empty square (to take it).
=============================================================*/
block_lose(State, Player, Place) :-
    next_player(Player, Opponent),
    fill_a_pair(State, Opponent, Place).

/*=============================================================
  next_player(+Player, -Opponent)
  next_player(atom, atom)
```

```
  The other player.
*************************************************************/
next_player(o, x).
next_player(x, o).

/*************************************************************
  fill_a_pair(+State, +Player, -Blank)
  fill_a_pair(list(atom), atom, integer)

  Look for a line in the board which contains two of the Player's
  tokens plus an empty square. If there is one, then
  return the position of the empty square.
*************************************************************/
fill_a_pair(State, Player, Blank) :-
    find_line(State, Line),        % generate a line
    pair_and_blank(Line, Player, Blank). % is there a pair?

/*************************************************************
  find_line(+State, -Line)
  find_line(list(atom),
            line(pair(atom,integer),
                 pair(atom,integer), pair(atom,integer)))

  Obtain each line in the board, in turn. Each line consists of
  three squares, and each square consists of the contents of the
  square (i.e. player token or empty) plus its numeric position
  in the board.
*************************************************************/
find_line([S1, S2, S3, S4, S5, S6, S7, S8, S9],
      line(sq(S1,1), sq(S2,2), sq(S3,3))).
find_line([S1, S2, S3, S4, S5, S6, S7, S8, S9],
      line(sq(S4,4), sq(S5,5), sq(S6,6))).
find_line([S1, S2, S3, S4, S5, S6, S7, S8, S9],
      line(sq(S7,7), sq(S8,8), sq(S9,9))).
find_line([S1, S2, S3, S4, S5, S6, S7, S8, S9],
      line(sq(S1,1), sq(S4,4), sq(S7,7))).
find_line([S1, S2, S3, S4, S5, S6, S7, S8, S9],
      line(sq(S2,2), sq(S5,5), sq(S8,8))).
find_line([S1, S2, S3, S4, S5, S6, S7, S8, S9],
      line(sq(S3,3), sq(S6,6), sq(S9,9))).
find_line([S1, S2, S3, S4, S5, S6, S7, S8, S9],
      line(sq(S1,1), sq(S5,5), sq(S9,9))).
find_line([S1, S2, S3, S4, S5, S6, S7, S8, S9],
      line(sq(S3,3), sq(S5,5), sq(S7,7))).

/*************************************************************
  pair_and_blank(+Line, +Player, -Place)
  pair_and_blank(line(atom-integer, atom-integer, atom-integer),
        atom, integer)

  Test whether line contains two tokens belonging to Player
```

```
  and an empty square. Return the position of the blank.
================================================================*/
pair_and_blank(line(sq(Player,_), sq(Player,_), sq(Empty,Blank)),
           Player, Blank) :-
    empty_square(Empty).
pair_and_blank(line(sq(Player,_), sq(Empty,Blank), sq(Player,_)),
           Player, Blank) :-
    empty_square(Empty).
pair_and_blank(line(sq(Empty,Blank), sq(Player,_), sq(Player,_)),
           Player, Blank) :-
    empty_square(Empty).

/*================================================================
  empty_square(+Square)
  empty_square(atom)

  Test if a square is empty
================================================================*/
empty_square(empty).

/*================================================================
  next_empty_square(+State, -Place)
  move(list(atom), integer)

  Find the position of the next empty square
================================================================*/

next_empty_square(State, Place) :-
    next_empty_square(State, [1,2,3,4,5,6,7,8,9], Place).

next_empty_square([Empty|_Squares], [Place|_], Place) :-
    empty_square(Empty).
next_empty_square([_Sq|Squares], [_|Places], Place) :-
    next_empty_square(Squares, Places, Place).

/*================================================================
  move(++move(Place, Token), +State, -State1)
  move(move(integer, atom), list(atom), list(atom))

  Move a player's Token to the Place'th square in State,
  giving State1
================================================================*/
move(move(Place, Token), State, State1) :-
    move(Place, Token, State, State1).

move(1, Token, [_Square|Squares], [Token|Squares]).
move(Place, Token, [Square|Squares], [Square|NewSquares]) :-
    Place > 1,
    P1 is Place-1,
    move(P1, Token, Squares, NewSquares).
```

Appendix D

Test Cases

The following figures describe the students' programs on which the system was tested. The system was tested on the student's entire program, but these are several hundred lines of code each and therefore they are too large to be included in full.

Each figure gives the code for the task of finding an empty square. To give an idea of the context, we have also included some of the surrounding code up to the level of the `choose_move/3` predicate required by the specification. Other clauses and predicates that form part of `choose_move/3` are omitted for reasons of space. Some of the predicates presented for context are not shown completely.

The programs are shown in their original state before any hand transformations have been performed and also in their final state, following the hand transformations, in the form in which they are identified.

Transformations: The following transformations were performed on these programs by hand, before the analysis.

Correct numbers of arguments that don't fit specification Some students changed the arguments to specified predicates so that they don't fit the specification. These were corrected.

Re-arrange arguments to match the order of synthesised arguments in predicate arguments and in terms used as data structures.

Fold and unfold predicates to make the predicate structure explicit. The test for an empty square typically had to be re-written as a separate predicate. It is occasionally also necessary (P7, P9) to group a sequence of calls togther into a call to a sub–predicate. This makes the predicate structure correspond to the strcuture that is synthesised from the task structure.

Divide 'or' branches into separate predicates.

Change lists into terms if students have used a Prolog list as a record (i.e. with a fixed size and elements of different types) then the list is changed to a Prolog term of the appropriate arity.

Remove extraneous control typically cuts.

Rewrite bugs Some buggy code has been corrected. P6 created a working program that was buggy due to an incorrect call. The bug caused problems for the type inference mechanism and was corrected.

Table D–1: Find empty square: P1

Domain Objects:	BOARD	EMPTY SQUARE	PLACE
Representation:	list of squares	variable	integer

Context:

```
choose_move(State,Host,[Place|Host]):-
    write('Thinking '),
    auto_choose(State,Host,Place,_,3), nl, !.

auto_choose([Turn,Lines,Board],
            Player,BestMove,BestScore,Ply):-
    bagof(Move,poss_move(Board,Move),Moves),
    bestmove([Turn,Lines,Board],
             Moves,BestMove,Player,Ply),
    make_and_eval(BestMove,[Turn,Lines,Board],
                  _,Player,BestScore).
```

Find Empty Square:

```
poss_move(Board,Place):-
    pick_nth(Place,Board,El),
    var(El).

pick_nth(1,[X|_],X).
pick_nth(N,[_|T],X):-
    pick_nth(N1,T,X),
    N is N1 + 1.
```

Table D–2: Find empty square: P2

Domain Objects:	BOARD	EMPTY SQUARE	PLACE
Representation:	nested term of lines	variable	x,y co-ordinates

Context:

```
choose_move(State, o_us, Move) :- !,
  need_to_try_move(o_us, State, Move),
  \+ (\+ (move(Move, State, New_state),
          cant_be_forced_to_lose(New_state, o_us))).

need_to_try_move(Player, State,
                 m(Some_coord, Player)) :-
    symmetries(State, Sym_list), !,
    co_ord(Some_coord),
    occupies(State, Some_coord, Nothing),
    var(Nothing),
    \+ (member(Orientation, Sym_list),
        pair(Orientation, Some_coord, Partner),
        precedes(Partner, Some_coord)).

occupies(board(Rows,_,_), X_coord/Y_coord, Player) :-
   Rowarg is X_coord + 1,
   Colarg is Y_coord + 1,
   arg(Rowarg, Rows, Line),
   arg(Colarg, Line, Player), !.
```

Find Empty Square:

It is difficult to identify any connected code that finds an empty square. Only part of the code is shown.

Table D–3: Find empty square: P3

Domain Objects:	BOARD	EMPTY SQUARE	PLACE
Representation:	list of lines	variable	not represented

Context:

```
choose_move(State,x) :- any_move(State,x),
```

Find Empty Square: Original

```
any_move([Line|_], Place) :-  any_space(Line,Place).
any_move([_|Lines], Place) :- any_move(Lines, Place).

any_space(line(Blank,_,_), Blank) :-
    var(Blank), Blank = Player, !.
any_space(line(_,Blank,_), Blank) :-
    var(Blank), Blank = Player, !.
any_space(line(_,_,Blank), Blank) :-
    var(Blank), Blank = Player, !.
```

This code actually makes the move, so it doesn't return a PLACE.

Find Empty Square: As Recognised

```
any_move([Line|_], Place) :-  any_space(Line,Place).
any_move([_|Lines], Place) :- any_move(Lines, Place).

any_space(line(Blank,_,_), Blank) :-
    empty_square(Blank).
any_space(line(_,Blank,_), Blank) :-
    empty_square(Blank).
any_space(line(_,_,Blank), Blank) :-
    empty_square(Blank).

empty_square(Square) :- var(Square).
```

Table D–4: Find empty square: P4

Domain Objects:	BOARD	EMPTY SQUARE	PLACE
Representation:	list of rows	integer	integer

Original Code:

```
choose_move(state(_,_,_,[Place|_],_),
            Player,
            move(Place, Player)).
```

Context: As recognised

```
choose_move(State, Player, move(Place, Player)) :-
    first_empty(State, Place).
```

Find Empty Square: As recognised

```
first_empty(State, First) :-
    empties(State, [First|_Empties]).

empties(state(_,_,_,Empties,_), Empties).
```

P4 maintains a list of empty squares as the fourth argument of `state/5`. So all that is necessary is to pick an element from the head of that list.

Table D–5: Find empty square: P5

Domain Objects:	BOARD	EMPTY SQUARE	PLACE
Representation:	list of lines	atom	integer

Context: As recognised

```
choose_move(State, Player, m(Position, Player)) :-
    i_win(State, Player, Place),
    block_lose(State, Player, Place),
    next_mv(State, Position).
```

Find Empty Square: Original

```
next_mv(State,Move) :- first_empty(State, Move, 0).

first_empty([H|State], Move, SoFar) :-
    Temp is SoFar + 1,
    ( (H = empty,
       Move is Temp)
    ;
     first_empty(State, Move, Temp)
    ).
```

Find Empty Square: As recognised

```
next_mv(State,Move) :- first_empty(State, 0, Move).

first_empty([H|State], SoFar, Move) :-
    Move is Sofar + 1,
    empty_square(H).
first_empty([_|State], SoFar, Move) :-
    Temp is Sofar + 1,
    first_empty(State,Temp,Move).

empty_square(empty).
```

Table D–6: Find empty square: P6

Domain Objects:	BOARD	EMPTY SQUARE	PLACE
Representation:	list of lines	integer	integer

Context:

```
choose_move(State, computer, [Position|o]):-
    computer_turn(State, Position).
computer_turn(State, Position):-
    avail(State, Position).
```

Find Empty Square: Original

```
avail([WinSet|_], Position) :- avail_winset(WinSet, Position).
avail([_|Rest], Position) :-   avail_winset(Rest, Position).

avail_winset([Position|_], Position) :- integer(Position), !.
avail_winset([_|Rest], Position):- avail_winset(Rest, Position).
```

Find Empty Square: As recognised

```
avail([WinSet|_], Position) :- avail(WinSet, Position).
avail([_|Rest], Position) :- avail_winset(Rest, Position).

avail_winset([Position|_], Position) :- empty_square(Position).
avail_winset([_|Rest], Position):- avail_winset(Rest, Position).

empty_square(N) :- integer(N).
```

Table D–7: Find empty square: P7

Domain Objects:	BOARD	EMPTY SQUARE	PLACE
Representation:	list of rows	atom	x,y co-ordinates

Context: Original

```
choose_move(State, computer, [R,C]):-
        getx(e, State, Epos),
        getx(x, State, Xpos),
        getx(o, State, Opos),
        set(WinStates),
        ((s_move(Opos, WinStates, Pos),
         \+Pos = [],
         find_common(Epos, Pos, H), !);
         (s_move(Xpos, WinStates, Pos),
         \+Pos = [],
         find_common(Epos, Pos, H), !);
        [H|T]=Epos),
        Row is (H-1) div 3,
        C is H - Row*3,
        row(R, Row).
```

Context: As recognised

```
choose_move(State, computer, [R,C]):-
    get_square(State, Place),
    Row is (Place-1) // 3,
    C is Place - Row*3,
    row(R, Row).

get_square(State, Place) :-
    empty(E),
    getx(E, State, EPos),
    set(WinStates),
    (    getx(o, State, OXpos)
    ;
         getx(x, State, OXpos)
    ),
    s_move(OXPos, WinStates, Pos),
    \+Pos = [],
    find_common(Epos, Pos, Place), !.
get_square(State, Place) :-
    empty_square(State, Place).
```

Table D–8: Find empty square: P7 continued

Find Empty Square: As recognised

```
empty_square(State, Place) :-
    empty(E),
    getx(E, State, [Place|_]).

empty(e).

getx(Mark, [R1, R2, R3], Pos_list):-
    belong(Mark, R1, 1, Row1),
    belong(Mark, R2, 4, Row2),
    belong(Mark, R3, 7, Row3),
    ap(Row1, Row2, Row3, Pos_list).

belong(_, [], _, []):- !.
belong(H, [H|T], Pos, [Pos|R]):- !,
    Pos1 is Pos + 1,
    belong(H, T, Pos1, R).
belong(X, [_|T], Pos, R):-
    Pos1 is Pos + 1,
    belong(X, T, Pos1, R).

ap(Xs, Ys, Zs, State):-
    append(Xs, Ys, Inter),
    append(Inter, Zs, State).
```

`getx/3`, `belong/4` and `ap/4` are the same as in the original code.

Table D–9: Find empty square: P8

Domain Objects:	BOARD	EMPTY SQUARE	PLACE
Representation:	list of squares	pair(integer, atom)	integer

Original Code:

```
choose_move(State, o, Move-o) :-
    s_2_board(State, Board),
    (win_move(Board, Move);
     block_move(Board, Move);
     other_move(Board, Move);
     member(Move-n, Move)),
    write('I move to position '),
    write(Move),nl,nl,
    !.

member(H, [H|_]).
member(I, [_|T]) :- member(I, T).
```

Context: As Recognised

```
choose_move(State, o, Move-o) :-
    s_2_board(State, Board),
    (win_move(Board, Move);
     block_move(Board, Move);
     other_move(Board, Move);
     find_empty_square(State, Move)),
    write('I move to position '),
    write(Move),nl,nl,
    !.
```

Find Empty Square: As Recognised

```
find_empty_square(State, Place) :-
    empty_square(Empty),
    member(State, Empty-Place).

empty_square(n).

member([H|_], H).
member([_|T], I) :- member(T, I).
```

Table D–10: Find empty square: P9

Domain Objects:	BOARD	EMPTY SQUARE	PLACE
Representation:	nested term of lines	variable	integer

Context:

```
choose_move(State,computer,Move/computer):-
    any_move(State,computer,Move),
    write('Computer has moved!'),nl,nl,!.
```

Find Empty Square: Original

```
any_move(State,_,Move):-
    all_empty_squares(9,State,List),!,
    permute([5,3,1,7,9,2,8,4,6],List,List1),
    member(Move,List1),
    empty_sq(Move,State).

all_empty_squares(0,_,[]).
all_empty_squares(N,Board,[N|List]):-
    empty_sq(N,Board),
    M is N - 1,
    all_empty_squares(M,Board,List).
all_empty_squares(N,Board,List):-
    \+ empty_sq(N,Board),
    M is N - 1,
    all_empty_squares(M,Board,List).

empty_sq(Board, N):-
   sortout(N,M1,M2),
   arg(M1, Rows, Anyline),
   arg(M2, Anyline, X),
   var(X).

sortout(N,M1,M2):- M2 is N mod 3, M2 \== 0, M1 is N div 3 + 1.
sortout(N,M1,3):-  M2 is N mod 3, M2 = 0,   M1 is N div 3.

member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

Table D–11: Find empty square: P9 continued

Find Empty Square: As recognised

```
any_move(State,_,Move):-
    all_empty_squares(9,State,List),!,
    permute([5,3,1,7,9,2,8,4,6],List,List1),
    member(Move,List1),
    empty_sq(Move,State).

empty_sq(Board, N):-
   square_and_pos(Board, N, X),
   empty_sq(X).

square_and_pos(board(Rows,_,_), N, X) :-
   sortout(N,M1,M2),
   arg(M1, Rows, Anyline),
   arg(M2, Anyline, X).

empty_sq(X) :- var(X).
```

Table D–12: Find empty square: P10

Domain Objects:	BOARD	EMPTY SQUARE	PLACE
Representation:	list of squares	atom	integer

Original Code:

```
choose_move(state(Board, _, Me), me, Me/Move) :-
    position_in(Board, *, Move).

position_in([H |_], H, 1).
position_in([_| T], Item, N) :-
        position_in(T, Item, N1),
        N is N1 + 1.
```

Context: As Recognised

```
choose_move(state(Board, _, Me), me, Me/Move) :-
    empty_position(Board, Move).
```

Find Empty Square: As recognised

```
empty_position(Board, Move) :-
    empty_square(Empty),
    position_in(Board, Empty, Move).

empty_square('*').

position_in([H |_], H, 1).
position_in([_| T], Item, N) :-
    position_in(T, Item, N1),
    N is N1 + 1.
```

Table D–13: Find empty square: P11

Domain Objects:	BOARD	EMPTY SQUARE	PLACE
Representation:	list of squares	atom	integer

Original Code:

```
choose_move('O',State,New_state) :-
    move('O',Posn,State,New_state),
    tactic('X',Posn,State).

move(Player, 1 , [' ',R2,R3,R4,R5,R6,R7,R8,R9],
                 [Player,R2,R3,R4,R5,R6,R7,R8,R9]).
move(Player, 2 , [R1,' ',R3,R4,R5,R6,R7,R8,R9],
                 [R1,Player,R3,R4,R5,R6,R7,R8,R9]).
....
move(Player, 9 , [R1,R2,R3,R4,R5,R6,R7,R8,' '],
                 [R1,R2,R3,R4,R5,R6,R7,R8,Player]).

tactic(Player,Posn,State) :-
    move(Player,Posn,State,State1),
    end_state(Player,_,State1).
```

Move and Assess: As Recognised

```
choose_move(State, 'O', Move) :-
        select_move(State, 'O', Move).

select_move(State, Player, move(Posn,Player)) :-
        move(move(Posn,Player), State, New_state),
        tactic(New_state, Player).

move(move(1, Player),
     [Empty,R2,R3,R4,R5,R6,R7,R8,R9],
     [Player,R2,R3,R4,R5,R6,R7,R8,R9]) :-
   empty_square(Empty).
move(move(2, Player),
     [R1,Empty,R3,R4,R5,R6,R7,R8,R9],
     [R1,Player,R3,R4,R5,R6,R7,R8,R9]) :-
    empty_square(Empty).
.......
move(move(9, Player),
     [R1,R2,R3,R4,R5,R6,R7,R8,Empty],
     [R1,R2,R3,R4,R5,R6,R7,R8,Player]) :-
    empty_square(Empty).

empty_square(' ').
```

Table D–14: Find empty square: P12

Domain Objects:	BOARD	EMPTY SQUARE	PLACE
Representation:	list of squares	atom	integer

Original Code

```
choose_move(State, Player, Player/Position) :-
    repeat,
    write('My turn'),nl,nl,
    write('Enter ''C'' to continue'),nl,nl,
    read(_),!,
    (((get_symbol(Player, Symbol)
      ;
       (next_player(Player, Opponent),
        get_symbol(Opponent, Symbol))
      ),
      may_win(State, Symbol, Choice))
    ;
     free_square(Choice, State)
    ).

free_square(Position, State) :-
    free_square(Position, [1,2,3,4,5,6,7,8,9], State).

free_square(Position, [H|_], [Position|_]) :-
    H = ' ',!.
free_square(Position, [_|T], [_|T1]) :-
    free_square(Position, T, T1).
```

Table D–15: Find empty square: P12 continued

Context: As recognised

```
choose_move(State, Player, m(Position, Player)) :-
    i_win(State, Player, Place) ;
    block_lose(State, Player, Place) ;
    free_square(State, Position).
```

Find Empty Square: As recognised

```
free_square(State, Position) :-
    free_square(State, [1,2,3,4,5,6,7,8,9], Position).

free_square([H|_], [Position|_], Position) :-
    empty_square(H).
free_square([_|T], [_|T1], Position) :-
    free_square(T, T1, Position).

empty_square(' ').
```

Table D–16: Find empty square: P13

Domain Objects:	BOARD	EMPTY SQUARE	PLACE
Representation:	list of squares	atom	integer

Context:

```
choose_move(State,computer,Move):-
    winning_move(State,Move);
    forced_move(State,Move);
    tactical_move(State,Move);
    any_move(State,Move).
```

Find Empty Square: Original

```
any_move(State, Result) :- any_move(State, 1, Result).

any_move([e|_],Result,Result):- !.
any_move([_|State],N,Result):-
    N1 is N + 1 ,
    any_move(State,N1,Result).

empty_square(e).
```

Find Empty Square: As recognised

```
any_move(State, Result) :- any_move(State, 1, Result).

any_move([H|_],Result,Result):- empty_square(H).
any_move([_|State],N,Result):-
    N1 is N + 1 ,
    any_move(State,N1,Result).

empty_square(e).
```

Table D–17: Find empty square: P14

Domain Objects:	BOARD	EMPTY SQUARE	PLACE
Representation:	list of squares	integer	integer

Original Code

```
choose_move(State, player(P, S), Move):-
    ( P = p, get_move(State, S, Move)) ;
    ( P = c, gen_move(State, S, Move)).

gen_move(State, S, Move):-
   tab(10), write('MY MOVE'), nl, find_move(State, S, Move).

find_move(State, S, move(Pos, S)):-
    move(move(Pos,S), State, State1),
    win_state(State1, player(c, S), c).
.....
find_move(State, S, move(Pos, S)):-
    move(move(Pos, S), State, State1),
    draw_state(State1, d).
```

Table D–18: Find empty square: P14 continued

Context: As recognised

```
choose_move(State, player(c, S), Move):-
    gen_move(State, player(c, S), Move).

gen_move(State, S, Move):-
   tab(10), write('MY MOVE'), nl, find_move(State, S, Move).
```

Move and Assess: As Recognised

```
find_move(State, player(Who, Token), move(Pos, Token)):-
   move(move(Pos, Token), State, State1),
   win_state(State1, player(Who, Token)).
......
find_move(State, player(_Who, Token), move(Pos, Token)):-
   move(move(Pos, Token), State, State1).

move(move(Pos, Token), [Pos|Tail], [Token|Tail]):-
   empty_square(Pos).
move(Move, [X|Tail], [X|Tail1]):-
   move(Move, Tail, Tail1).

empty_square(Pos) :- integer(Pos).
```

Table D–19: Find empty square: P15

Domain Objects:	BOARD	EMPTY SQUARE	PLACE
Representation:	term of squares	atom	integer

Context:

```
choose_move(State, [c/C, _/E], Move/C) :-
  c_choose_move(State, Move/C/E), !.

c_choose_move(State, Move/C/E) :-
  nl, write('It is the computer''s turn to move'), nl,
  write('It is playing  '), write(C), nl,
  think(State, Move/C/E).

think(State, Move/C/_) :-
  winning_position(State, Move/C), !.
...
think(State, Move/_/_) :-
  pick_move(State, Move).
```

Find Empty Square: Original

```
pick_move(State, Move) :-
    (Move = 1; Move = 9; Move = 7; Move = 3;
     Move = 2; Move = 4; Move = 6; Move = 8; Move = 5
    ),
    legal_move(State, Move).

legal_move(State, Move) :-
    integer(Move),
    Move >= 1,
    Move =< 9,
    arg(Move, State, e).
```

Table D–20: Find empty square: P15 continued

Find Empty Square: As recognised

```
pick_move(State, Move) :-
    best_move(Move),
    legal_move(State, Move).

best_move(1).
best_move(9).
....
best_move(5).

legal_move(State, Move) :-
    get_square(State, Move, Sq),
    empty_square(Sq).

get_square(State, Move, Sq) :- arg(Move, State, Sq).

empty_square(e).
```

Non-recursive solution. Generates each possible PLACE in turn, in a preferred sequence, and tests that the PLACE is empty. If not, it fails back to get another PLACE in the sequence.

Table D–21: Find empty square: P16

Domain Objects:	BOARD	EMPTY SQUARE	PLACE
Representation:	list of squares	integer	integer

Original Code:

```
choose_move(State,computer,Move):-
   good_move(State,o,Move).

good_move(State,o,[Number,o]):-
   move([Number,o],State,State1),
   win_state(State1,computer,computer).
....
good_move(State,o,[Number,o]):-
   move([Number,o],State,State1),
   member(Number,[2,4,6,8]).

move([_,_],[],[]).
move([Number,Symbol],[Number|Tail],[Symbol|Tail]):-
    integer(Number).
move([Number,Symbol],[X|Tail],[X|NewTail]):-
    move([Number,Symbol],Tail,NewTail).
```

P16's code for move/3 and its use to find an empty square is identical to P14.

`move([-Place, +Token], +List, -NewList)` returns the first element that fulfills some condition (i.e. is an integer) and replaces that element with the Token to give the NewList.

Table D–22: Find empty square: P16 continued

Context: As recognised

```
choose_move(State,computer,Move):-
        good_move(State,o,Move).
```

Move and Assess: As Recognised

```
good_move(State, Token, move(Number,Token)):-
        move(move(Number,Token), State, State1),
        win_state(State1, Token).
....
good_move(State, Token, move(Number, Token)):-
        move(move(Number, Token), State, State1).

move(move(Number,Symbol),
     [Number|Tail], [Symbol|Tail]):-
        empty_square(Number).
move(Move, [X|Tail], [X|NewTail]):-
        move(Move, Tail, NewTail).

empty_square(Number) :- integer(Number).
```

Appendix E

Tasks and Prototypical Predicates

This Appendix shows the definitions of tasks and prototypes that are used for recognising the noughts and crosses program. It is divided into three parts. The first part shows the tasks and prototypes which match the entire demonstration program whose code is shown in Appendix B. This section shows prototypes and task definitions for a range of different tasks which between them implement a large part of the noughts and crosses program. It does not show the variations that different implementors may create for the same task.

The second part shows the range of task structures and prototypes that are necessary to represent the different variants of a single quite small task as implemented by different students. The task is that of finding an empty square, and the task definition and prototypes are derived from and account for the 16 students' implementations in Appendix D.

The final part shows one domain–independent task and its prototype, which was identified in several of the students' programs.

E.1 Noughts and Crosses

This section shows the definitions of tasks and prototypes that were used to recognise the noughts and crosses program in Appendix B and which is discussed in Section 6.3.

The prototypes are mainly task specific rather than being general. One prototype is shown for each task.

Table E–1: Noughts and Crosses Task: *play_game/1*

Sub–tasks:	
Role	Body
initialise/2	*initialise*/2
display_game/2	*display_game*/2
end_state/2	*end_state*/2
announce/1	*announce*/1
choose_move/3	*choose_move*/3
move/3	*move*/3

Game objects:	RESULT

The top–level task for all the noughts and crosses games.

Table E–2: Noughts and Crosses Prototype: GAME_TOP

Prototype:	GAME_TOP	
Top predicate:	play/1	
Prototype functors:	play/1	play/3

```
play(Result) :- initialise(State, Player),
                display_game (State, Player),
                play(State, Player, Result).

play(State, _, Result) :-
                end_state(State, Result),
                !,
                announce(Result).
play(State, Player, Result) :-
                choose_move (State, Player, Move),
                move (Move, State, State1),
                display_game (State1, Player),
                next_player (Player, Player1),
                !,
                play(State1, Player1, Result).
```

The top–level prototype for all the noughts and crosses games.

Table E–3: Noughts and Crosses Task: *next_player/2*

Sub-tasks:	
Role	Body
assign_A(A)	`A = '$tokenA'`
assign_B(B)	`B = '$tokenB'`

Game objects:	PLAYER

Exchange players.
The items starting with a `'$'` symbol can match any Prolog term, so this matches any representation for the PLAYER.
We only provide one prototype and one task breakdown for this task, so as to be sufficient for the demonstration example.

Table E–4: Noughts and Crosses Prototype: EXCHANGE_TWO

Prototype:	EXCHANGE_TWO
Top predicate:	`exchange_two/2`
Prototype functors:	`exchange_two/2`

```
exchange_two(First, Second) :- assign_A(First),
                               assign_B(Second).
exchange_two(First, Second) :- assign_B(First),
                               assign_A(Second).
```

Prototype to swap one argument with another. Expected use is only for *next_player*.

Table E–5: Noughts and Crosses Task: *move/3*

Sub-tasks:	
Role	Body
unwrap(*Move*, *Place*, *Player*)	`Move = pair(Place, Player)`

Game objects:	MOVE	BOARD

Make a move.
Sufficient for the demonstrations but simpler than the students' versions.

Table E–6: Noughts and Crosses Prototype: REPLACE_NTH

Prototype:	REPLACE_NTH	
Top predicate:	`replace_nth/3`	
Prototype functors:	`replace_nth/3`	`replace_nth/4`

```
replace_nth(Input, List, NewList) :-
            unwrap(Input, Position, NewEntry),
            replace_nth(Position, NewEntry, List, NewList).
replace_nth(1, NewEntry, [_|T], [NewEntry|T]) :- true.
replace_nth(Position, NewEntry, [H|T], [H|New]) :-
            Position > 1,
            P1 is Position - 1,
            replace_nth(P1, NewEntry, T, New).
```

This is a variant of Gegg-Harrison's Schema E.
This is a highly instantiated prototype, with the initialisation of the counter, decrement and step fixed instead of being left to the task specification. It also has an extra level of unwrapping the pair into separate elements, which strictly speaking is also rather task specific.

Table E–7: Noughts and Crosses Task: *choose_move/3*

Sub-tasks:	
Role	Body
i_win/3	*i_win*/3
block_lose/3	*block_lose*/3
find_empty_square/2	*find_empty_square*/2
build_structure(*Place*, *Player*, *Move*)	`Move = move(Place, Player)`
move/3	*move*/3
win_for_player/2	*win_for_player*/2
next_player/2	*next_player*/2
heuristic_evaluation/2	*heuristic_evaluation*/2

Game objects:	MOVE	BOARD

Choose a move.
Demonstration version and also some students' versions.

Table E–8: Noughts and Crosses Prototype: CHOOSE_MOVE_HEURISTIC

Prototype:	CHOOSE_MOVE_HEURISTIC
Top predicate:	`choose_move/3`
Prototype functors:	`choose_move/3`

```
choose_move(State, Player, Move) :-
      (  i_win(State, Player, Place)
      ;  block_lose (State, Player, Place)
      ;  find_empty_square (State, Place)),
      build_structure (Place, Player, Move).
```

A prototype for heuristically choosing the move. Leaves out the user's move. Looks at the current state and decides which move to make.

Table E–9: Noughts and Crosses Prototype: MOVE_AND_ASSESS

Prototype:	MOVE_AND_ASSESS
Top predicate:	choose_move/3
Prototype functors:	choose_move/3

```
choose_move(State, Player, Move) :-
          Move=move(_Place, Player),
          move (Move, State, State1),
          win_for_player (State1, Player).
choose_move(State, Player, Move) :-
          next_player (Player, Opponent),
          OppMove=move(Place, Opponent),
          move (OppMove, State, State1),
          win_for_player (State1, Opponent).
choose_move(State, Player, Move) :-
          Move=move(_Place, Player),
          move (Move, State, State1),
          heuristic_evaluation (State1, Player).
choose_move(State, Player, Move) :-
          Move=move(_Place, Player),
          move (Move, State, _State1).
```

A prototype for heuristically choosing the move. Leaves out the user's move. Tries different moves in turn. Creates a new state in which the move is made, then evaluates the result of making that move.
This approach was used in students' code: P11 P14 P16

Table E–10: Noughts and Crosses Task: *i_win*

Sub-tasks:	
Role	Body
fill_a_pair/3	*line_next_move*/3

Complete a line.

Table E–11: Noughts and Crosses Prototype: WIN_IN_ONE

Prototype:	WIN_IN_ONE
Top predicate:	`i_win/3`
Prototype functors:	`i_win/3`

```
i_win(State, Player, Place) :- fill_a_pair(State, Player, Place).
```

Win if computer can make a move between two tokens.

Table E–12: Noughts and Crosses Task: *block_lose*

Sub–tasks:	
Role	Body
fill_a_pair/3	*line_next_move*/3
next_player/2	*next_player*/2

Block opponent's line.

Table E–13: Noughts and Crosses Prototype: BLOCK_LOSE_NEXT_MOVE

Prototype:	BLOCK_LOSE_NEXT_MOVE
Top predicate:	`block_lose/3`
Prototype functors:	`block_lose/3`

```
block_lose(State, Player, Place) :- next_player (Player, Opponent),
                                    fill_a_pair(State, Player, Place).
```

Block if computer can make a move between opponent's two tokens.

Table E–14: Noughts and Crosses Task: *line_next_move*

Sub-tasks:	
Role	Body
find_line/2	*find_line*/2
pair_and_blank/3	*pair_and_blank*/3

Game objects:	BOARD	PLAYER	PLACE

Find two tokens plus a blank in a line and fill the gap.

Table E–15: Noughts and Crosses Prototype: FILL_A_PAIR_IN_LINES

Prototype:	FILL_A_PAIR_IN_LINES
Top predicate:	`fill_a_pair/3`
Prototype functors:	`fill_a_pair/3`

```
fill_a_pair([Line|_State], Player, Blank) :-
                    pair_and_blank (Line, Player, Blank).
fill_a_pair([_Line|State], Player, Blank) :-
                    fill_a_pair(State, Player, Blank).
```

Find a pair and fill the gap — recursively, in a list of lines

Table E–16: Noughts and Crosses Prototype: FILL_A_PAIR_IN_SQUARES

Prototype:	FILL_A_PAIR_IN_SQUARES
Top predicate:	`fill_a_pair/3`
Prototype functors:	`fill_a_pair/3`

```
fill_a_pair(State, Player, Blank) :-
                    find_line (State, Line),
                    pair_and_blank (Line, Player, Blank).
```

Find a pair and fill the gap — in a list of squares, construct each line, then test it.

Table E–17: Noughts and Crosses Task: *pair_and_blank/3*

Sub–tasks:	
Role	Body
empty_square/1	*empty_square*/1
square_contains_players/2	*square_contains_players*/2

Game objects:	LINE	PLAYER	SQUARE (PLACE?)

Test for a pair of tokens and a blank square in a line.

Table E–18: Noughts and Crosses Prototype: PAIR_AND_BLANK

Prototype:	PAIR_AND_BLANK
Top predicate:	`pair_and_blank/3`
Prototype functors:	`pair_and_blank/3`

```
pair_and_blank(line(sq(Player,_),sq(Player,_),sq(Empty,Blank)),
               Player, Blank) :-
               empty_square (Empty).
pair_and_blank(line(sq(Player,_),sq(Empty,Blank),sq(Player,_)),
               Player, Blank) :-
               empty_square (Empty).
pair_and_blank(line(sq(Empty,Blank),sq(Player,_),sq(Player,_)),
               Player, Blank) :-
               empty_square (Empty).
```

Analyse a line to see if it consists of two players and a token.
Version assumes that each line consists of a term with three squares,
and that each square is a pair of a token and a position.

Table E–19: Noughts and Crosses Prototype: PAIR_AND_BLANK_VAR

Prototype:	PAIR_AND_BLANK_VAR
Top predicate:	pair_and_blank/3
Prototype functors:	pair_and_blank/3

```
pair_and_blank(line(Sq1, Sq2, Sq3), Player, Blank) :-
          square_contains_players(Sq1, Player),
          square_contains_players(Sq2, Player),
          empty_square (Sq3),
          unify(Blank, Sq3).
pair_and_blank(line(Sq1, Sq2, Sq3), Player, Blank) :-
          square_contains_players(Sq1, Player),
          square_contains_players(Sq3, Player),
          empty_square (Sq2),
          unify(Blank, Sq2).
pair_and_blank(line(Sq1, Sq2, Sq3), Player, Blank) :-
          square_contains_players(Sq2, Player),
          square_contains_players(Sq3, Player),
          empty_square (Sq1),
          unify(Blank, Sq1).
```

Analyse a line to see if it consists of two players and a token.
This variant requires that each line consists of a term with three squares, and that each square is either a variable for an empty square or else a token for a player.

Table E–20: Noughts and Crosses Task: *find_line/2*

Sub-tasks:	
Role	Body
$fl1(A, B)$	`A=[S1, S2, S3, _S4, _S5, _S6, _S7, _S8, _S9],` `B=line(sq(S1,1), sq(S2,2), sq(S3,3))`
$fl2(A, B)$	`A=[_S1, _S2, _S3, S4, S5, S6, _S7, _S8, _S9],` `B=line(sq(S4, 4), sq(S5,5), sq(S6,6))`
$fl3(A, B)$	`A=[_S1, _S2, _S3, _S4, _S5, _S6, S7, S8, S9],` `B=line(sq(S7,7), sq(S8,8), sq(S9,9))`
$fl4(A, B)$	`A=[S1, _S2, _S3, S4, _S5, _S6, S7, _S8, _S9],` `B=line(sq(S1,1), sq(S4,4), sq(S7,7))`
$fl5(A, B)$	`A=[_S1, S2, _S3, _S4, S5, _S6, _S7, S8, _S9],` `B=line(sq(S2,2), sq(S5,5), sq(S8,8))`
$fl6(A, B)$	`A=[_S1, _S2, S3, _S4, _S5, S6, _S7, _S8, S9],` `B=line(line(sq(S3,3), sq(S6,6), sq(S9,9))`
$fl7(A, B)$	`A=[S1, _S2, _S3, _S4, S5, _S6, _S7, _S8, S9],` `B=line(sq(S1,1), sq(S5,5), sq(S9,9))`
$fl8(A, B)$	`A=[_S1, _S2, S3, _S4, S5, _S6, S7, _S8, _S9],` `B=line(sq(S3,3), sq(S5,5), sq(S7,7))`

Game objects:	BOARD	LINE

Build up a line in a list of squares by pattern matching.

Table E–21: Noughts and Crosses Prototype: FIND_LINE_IN_SQUARES

Prototype:	FIND_LINE_IN_SQUARES
Top predicate:	`find_line_in_squares/2`
Prototype functors:	`find_line_in_squares/2`

```
find_line_in_squares(State, Line) :- fl1 (State, Line).
find_line_in_squares(State, Line) :- fl2 (State, Line).
find_line_in_squares(State, Line) :- fl3 (State, Line).
find_line_in_squares(State, Line) :- fl4 (State, Line).
find_line_in_squares(State, Line) :- fl5 (State, Line).
find_line_in_squares(State, Line) :- fl6 (State, Line).
find_line_in_squares(State, Line) :- fl7 (State, Line).
find_line_in_squares(State, Line) :- fl8 (State, Line).
```

Build up a line in a list of squares by pattern matching.

Table E–22: Noughts and Crosses Task: *empty_square/1*

Sub-tasks:	
Role	Body
test(*Square*)	`Square = '$empty_square'`

Game objects:	SQUARE

Test a square to see if it's empty.

Table E–23: Noughts and Crosses Prototype: ONE_OFF_TEST

Prototype:	ONE_OFF_TEST
Top predicate:	`one_off_test/1`
Prototype functors:	`one_off_test/1`

```
one_off_test(Item) :- test(Item).
```

Test an item by calling a predicate.
Ideally we might like some further constraints on what this could match. In particular to distinguish between a test and a generator, we might want mode annotations that insist that the `Item` is instantiated when the predicate `one_off_test/1` is called.

Table E–24: Noughts and Crosses Task: *find_empty_square/2*

Sub–tasks:	
Role	Body
base(*Base*)	`Base = 1`
increment(*This*, *Next*)	`Next is This + 1`
test(*Square*)	*empty_square* `(Square)`

Game objects:	BOARD	SQUARE

Look for an empty square.
This is the version required for the demonstration program. The versions for the students' programs are presented later in this Appendix.

Table E–25: Noughts and Crosses Prototype: TESTED_ELEMENT_POSITION_ACC

Prototype:	TESTED_ELEMENT_POSITION_ACC
Top predicate:	`element_position/2`
Prototype functors:	`element_position/2` `element_position/3`

```
element_position(List, Result) :-
                    base(Base),
                    element_position(List, Base, Result).

element_position([H|_T], Base, Base) :-
                    test (H).
element_position([_H|T], Acc, Result) :-
                    increment (Acc, Next),
                    element_position(T, Next, Result).
```

Find the position of the element in a list that satisfies some test.
Related to Gegg–Harrison's Schema C.

E.2 The Task of finding an Empty Square

This section shows the range of task structures and prototypes that are necessary to represent the different variants of a single quite small task as implemented by different students. The task is that of finding an empty square, and the task definition and prototypes are derived from and account for the 16 students' implementations in Appendix D.

The first table shows the task definition for finding an empty square. This has many different sub-tasks. In this table, the groups of sub-tasks used by each student are show separately for clarity. In the system's task definition, each sub-task appears exactly once and all the sub-tasks for the task are grouped together without any reference to the prototypes in which they might appear. A sub-task can appear in any prototype that refers to that role.

After that, the prototypes for each implementation are shown. They vary from the task specific to the very general.

The move and assess approach is shown separately. It is represented by a different task structure so its tasks do not appear with those for find empty square.

Also shown are the task and prototype definitions for a sub-task of the task of finding an empty square. This task tests a square to see if it is empty.

The task of finding an empty square was not identified for P2. The find empty square task was identified as such for P1 P3 P4 P5 P6 P7 P8 P9 P10 P12 P13 P15. The move and assess approach to finding an empty square was identified for P11 P14 P16.

We have found an approximate grouping of implementations based on the data representation, although there are a variety of exceptions. There is a connection between the data structure chosen for the board and the general approach taken to the implementation and the use of recursive or non-recursive techniques, as shown by table 6-11. Further study of more students' programs is needed to show

which implementations were typical and which are exceptional, and to show the full range of possible implementations.

Table E–26: Find Empty Square: *find_empty_square*

Used by:	Sub-tasks: Role:	Body: Prolog Code or *sub-task specification*
P9	*set_index*(*Index*)	`Index = 9`
	collect/3	*list_of_empty_squares*/3
	set_criteria(*Order*)	`Order = [5,3,1,7,9,2,8,4,6]`
	sort/3	*sort*/3
	pick/2	*pick*/2
P15	*match*/1	*pick_any_square*/1
	test/2	*empty_square*/2
P4	*get_list_from_record*/2	*access_empties*/2
	get_one_from_list(*Empties*, *Empty*)	`Empties=[Empty|_Rest]`
P7	*get_components*(*Struct*, *S1*, *S2*, *S3*)	`Struct=[S1,S2,S3]`
	get_indices(*I1*, *I2*, *I3*)	`(I1 = 1, I2 = 4, I3 = 7)`
	join_lists/4	*join_lists*/4
	increment(*This*, *Next*)	`Next is This + 1`
	test(*Square*)	*empty_square* (Square)
P1 P5 P10	*base*(*Base*)	`Base = 1`
P12 P13	*increment*(*This*, *Next*)	`Next is This + 1`
	test(*Square*)	*empty_square* (Square)
P3 P6 P8	*test*(*Square*)	*empty_square* (Square)

Game objects:	BOARD	PLACE

Find an empty square.

Table E–27: Find Empty Square Prototype: COLLECT_AND_SORT

Prototype:	COLLECT_AND_SORT
Top predicate:	`collect_and_sort/2`
Prototype functors:	`collect_and_sort/2`

```
collect_and_sort(Structure, Element) :-
                    set_index (Index),
                    collect (Structure, Index, List),
                    set_criteria (Criteria),
                    sort (Criteria, List, Sorted),
                    pick (Sorted, Element).
```

Used by P9.
Initialise an index to a structure: use it to collect all elements in a structure; sort the list of elements according to some criteria; pick the top element from the sorted list.

Table E–28: Find Empty Square Prototype: MATCH_AND_TEST

Prototype:	MATCH_AND_TEST
Top predicate:	`match_and_test/2`
Prototype functors:	`match_and_test/2`

```
match_and_test(Structure, Index) :-
                    match (Index),
                    test (Structure, Index).
```

Used by P15.
Generate an index, then test the element at that position in the structure.

Table E–29: Find Empty Square Prototype: GET_ONE_FROM_LIST_IN_RECORD

Prototype:	GET_ONE_FROM_LIST_IN_RECORD
Top predicate:	`get_one_from_list_in_record/2`
Prototype functors:	`get_one_from_list_in_record/2`

```
get_one_from_list_in_record(Record, Item) :-
                    get_list_from_record (Record, List),
                    get_one_from_list (List, Item).
```

Used by P4

Table E–30: Find Empty Square Prototype: GATHER_POSITIONS

Prototype:	GATHER_POSITIONS
Top predicate:	gather_positions/2
Prototype functors:	gather_positions/2 gather_positions/3 gather_positions/4

```
gather_positions(Struct, Pos) :-
                    test (Item),
                    gather_positions(Item, Struct, PosList),
                    get_one_from_list (List, Item).
gather_positions(Item, Struct, PosList) :-
                    get_components (Struct, S1, S2, S3),
                    get_indices (I1, I2, I3),
                    gather_positions(Item, S1, I1, Pos1),
                    gather_positions(Item, S2, I2, Pos2),
                    gather_positions(Item, S3, I3, Pos3),
                    join_lists (Pos1, Pos2, Pos3, PosList).
gather_positions(_Item, [], _, []) :- !.
gather_positions(Item, [Item|Ins], Out, [Out|Outs]) :-
                    !,
                    increment(Out, Next),
                    gather_positions(Item, Ins, Next, Outs).
gather_positions(Item, [_Reject|Ins], Out, Outs) :-
                    increment(Out, Next),
                    gather_positions(Item, Ins, Next, Outs).
```

Pick components out of a structure. Used by P7.

Table E–31: Find Empty Square Prototype: MATCHED_ELEMENT_POSITION

Prototype:	MATCHED_ELEMENT_POSITION
Top predicate:	`element_position/2`
Prototype functors:	`element_position/2` `element_position/3`

```
element_position(Structure, Position) :-
                    test (Item),
                    element_position(Structure, Item, Position).

element_position([Item|_T], Item, Base) :-
                    base (Base).
element_position([_H|T], Item, Position) :-
                    element_position(T, Item, Prev),
                    increment (Prev, Position).
```

Used by P10.
Find the position of one element that matches in a list of elements.
This prototype is naive, i.e. not tail recursive – we also have a tail recursive prototype but no–one used it.
Related to Gegg–Harrison's Schema C.

Table E–32: Find Empty Square Prototype: TESTED_ELEMENT_POSITION_ACC

Prototype:	TESTED_ELEMENT_POSITION_ACC
Top predicate:	`element_position/2`
Prototype functors:	`element_position/2` `element_position/3`

```
element_position(List, Result) :-
                    base(Base),
                    element_position(List, Base, Result).
element_position([H|_T], Base, Base) :-
                    test (H).
element_position([_H|T], Acc, Result) :-
                    increment (Acc, Next),
                    element_position(T, Next, Result).
```

Used by P5 P12 P13.
Find the position of one element that satisfies some test in a list of elements.
This prototype is tail recursive – we also have a version that wasn't but no–one used it.
Related to Gegg–Harrison's Schema C.

Table E–33: Find Empty Square Prototype: TESTED_ELEMENT_POSITION_GT

Prototype:	TESTED_ELEMENT_POSITION_GT
Top predicate:	`element_position/2`
Prototype functors:	`element_position/2` `element_position/3`

```
element_position(List, Pos) :-
                    element_position(List, Item, Pos),
                    test (Item).
element_position([H|_T], H, Base) :-
                    base (Base).
element_position([_H|T], Item, Pos) :-
                    element_position(T, Item, Prev),
                    increment (Prev, Pos),
```

Used by P1.
Find the position of one element that satisfies some test in a list of elements. A generate–and–test version with a naive (not tail recursive)implementation.
Related to Gegg–Harrison's Schema C.

Table E–34: Find Empty Square Prototype: NESTED_RECURSIVE_TEST

Prototype:	NESTED_RECURSIVE_TEST
Top predicate:	`nested_recursive_test/2`
Prototype functors:	`nested_recursive_test/2` `nested_recursive_test_sub/2`

```
nested_test([H|_L], Element) :-
                nested_test_sub(H, Element).
nested_test([_H|L], Element) :-
                nested_test(L, Element).

nested_test_sub([H|L], H) :-
                test (H).
nested_test_sub([_|L], Element) :-
                nested_test_sub(L, Element).
```

Used by P6.
Pick out one element (square) from a recursive structure (board) that contains recursive elements (lines).
Related to Gegg–Harrison's Schema C.

Table E–35: Find Empty Square Prototype: NESTED_TEST

Prototype:	NESTED_TEST	
Top predicate:	`nested_test/2`	
Prototype functors:	`nested_test/2`	`nested_test_sub/2`

```
nested_test([H|_L], Element) :- nested_test_sub(H, Element).
nested_test([_H|L], Element) :- nested_test(L, Element).

nested_test_sub(triple(E1,_E2,_E3), E1) :- test (E1).
nested_test_sub(triple(_E1,E2,_E3), E2) :- test (E2).
nested_test_sub(triple(_E1,_E2,E3), E3) :- test (E3).
```

Used by P3.
Pick out one element (square) from a recursive structure that contains triples. The outer structure (board) is a list, the inner structure (line) is a term.
Related to Gegg–Harrison's Schema C.

Table E–36: Find Empty Square Prototype: LOOKUP_MEMBER

Prototype:	LOOKUP_MEMBER	
Top predicate:	`set_lookup/2`	
Prototype functors:	`set_lookup/2`	`lookup/2`

```
set_lookup(L, Item) :-
                test (Index),
                lookup(List, pair(Index,Item)).

lookup([Pair|_L], Pair) :- true.
lookup([_| List], Pair) :- lookup(List, Pair).
```

Used by P8.
Look up an indexed item in a list of pairs.

Table E–37: Find Empty Square Sub-Task: *list of empty_squares/3*

Used by:	Sub-tasks:	Prolog Code or called sub tasks
P9	*base*(*Index*)	`Index = 0`
	decrement(*Index*, *Next*)	`Next is Index - 1`
	test/2	*empty_square*/2

Create a list of empty squares.

Table E–38: Find Empty Square Sub Prototype: LIST_OF_TESTED_INDICES

Prototype:	LIST_OF_TESTED_INDICES
Top predicate:	`list_of_tested_indices/3`
Prototype functors:	`list_of_tested_indices/3`

```
list_of_tested_indices(_Structure, Index, []) :-
                    base (Index).
list_of_tested_indices(Structure, Index, [Index|T]) :-
                    test (Structure, Index),
                    !,
                    decrement (Index, Next),
                    list_of_tested_indices(Structure, Next, T).
list_of_tested_indices(Structure, Index, T) :-
                    decrement (Index, Next),
                    list_of_tested_indices(Structure, Next, T).
```

Used by P9.

Table E–39: Find Empty Square Sub–Task: *empty_square/2*

Used by:	Sub–tasks:	**Prolog Code** or called sub–tasks
P9 P15	*index_struct*/3	*access_square*/3
	i_test/1	*empty_square*/1

Get an empty square in a complex board structure.

Table E–40: Find Empty Square Sub Prototype: INDEX_AND_TEST

Prototype:	INDEX_AND_TEST
Top predicate:	**index_and_test/2**
Prototype functors:	**index_and_test/2**

```
index_and_test(Structure, Index) :-
                    index_struct (Struct, Index, Element),
                    i_test (Element).
```

Used by P9, P15.
Obtain an indexed item from a complex structure.

Table E–41: Find Empty Square Sub–Task: *pick_any_square1*

Used by:	Roles:	Prolog Code or called sub–tasks
P9	*picked_item_of_nine_1*(*Pos*)	`Pos = '$position_1'`
	picked_item_of_nine_2(*Pos*)	`Pos = '$position_2'`
	picked_item_of_nine_3(*Pos*)	`Pos = '$position_3'`
	picked_item_of_nine_4(*Pos*)	`Pos = '$position_4'`
	picked_item_of_nine_5(*Pos*)	`Pos = '$position_5'`
	picked_item_of_nine_6(*Pos*)	`Pos = '$position_6'`
	picked_item_of_nine_7(*Pos*)	`Pos = '$position_7'`
	picked_item_of_nine_8(*Pos*)	`Pos = '$position_8'`
	picked_item_of_nine_9(*Pos*)	`Pos = '$position_9'`

Get an empty square in a complex board structure.

Table E–42: Find Empty Square Sub Prototype: PICK_NINE

Prototype:	PICK_NINE
Top predicate:	`pick_nine/1`
Prototype functors:	`pick_nine/1`

```
pick_nine(Item) :- picked_item_of_nine_1 (Item).
pick_nine(Item) :- picked_item_of_nine_2 (Item).
pick_nine(Item) :- picked_item_of_nine_3 (Item).
pick_nine(Item) :- picked_item_of_nine_4 (Item).
pick_nine(Item) :- picked_item_of_nine_5 (Item).
pick_nine(Item) :- picked_item_of_nine_6 (Item).
pick_nine(Item) :- picked_item_of_nine_7 (Item).
pick_nine(Item) :- picked_item_of_nine_8 (Item).
pick_nine(Item) :- picked_item_of_nine_9 (Item).
```

Used by P9.
Return an item with 9 options.

Table E–43: Find Empty Square Sub–Task: *access empties/1*

Used by:	Sub–tasks:	`Prolog Code` or called sub–tasks
P4	*match*(*Item*, *Out*)	`Item = Out`

Create a list of empty squares.

Table E–44: Find Empty Square Sub Prototype: ACCESS_FOUR_OF_FIVE

Prototype:	ACCESS_FOUR_OF_FIVE
Top predicate:	`access_four_of_five/2`
Prototype functors:	`access_four_of_five/3`

```
access_four_of_five(quin(_One, _Two, _Three, Four, _Five),
                    Out) :-
                  match (Four, Out).
```

Used by P4.

Table E–45: Test a Square Task: *empty_square/1*

Sub-tasks:	
Role	Body
test(*Square*)	`Square = '$empty_square'` VAR_TEST INTEGER_TEST

Game objects:	SQUARE

Test a square to see if it's empty.

Table E–46: Test Prototype: ONE_OFF_TEST

Prototype:	ONE_OFF_TEST
Top predicate:	`one_off_test/1`
Prototype functors:	`one_off_test/1`

```
one_off_test(Item) :- test(Item).
```

Test an item by calling a predicate.
Used to find an empty square by P4 P5 P7 P8 P10 P11 P12 P13

Table E–47: Test Prototype: VAR_TEST

Prototype:	VAR_TEST
Top predicate:	`var_test/1`
Prototype functors:	`var_test/1`

```
var_test(Item) :- var(Item).
```

Test to see if an item is an uninstantiated variable.
Used to find an empty square by P3.

Table E–48: Test Prototype: INTEGER_TEST

Prototype:	INTEGER_TEST
Top predicate:	`integer_test/1`
Prototype functors:	`integer_test/1`

```
integer_test(Item) :- integer(Item).
```

Test if an item is an integer.
Used to find an empty square by P6 P14 P16.

Table E–49: Move and Assess Approach Task: *move/3*

Used by:	Sub-tasks:	**Prolog Code** or called sub-tasks
	test/1	*empty_square*/1

If a square is empty we can take it.
Used by P11 P14 P16.

Table E–50: Move and Assess Approach Prototype: REPLACE_TEST_OR_MATCH

Prototype:	REPLACE_TEST_OR_MATCH
Top predicate:	**replace/3**
Prototype functors:	**replace/3**

```
replace(pair(H, New), [H|T], [New|T]) :-
                    test (H).
replace(Pair, [H|T], [H|Rest]) :-
                    replace(Pair, T, Rest).
```

Recursive replace used by P14 P16.

Table E–51: Move and Assess Approach Prototype: REPLACE_NONREC_NINE

Prototype:	REPLACE_NONREC_NINE
Top predicate:	`replace/3`
Prototype functors:	`replace/3`

```
replace(pair(1, New),
        [Old,S2,S3,S4,S5,S6,S7,S8,S9],
        [New,S2,S3,S4,S5,S6,S7,S8,S9]) :- test (Old).
replace(pair(2, New),
        [S1,Old,S3,S4,S5,S6,S7,S8,S9],
        [S1,New,S3,S4,S5,S6,S7,S8,S9]) :- test (Old).
replace(pair(3, New),
        [S1,S2,Old,S4,S5,S6,S7,S8,S9],
        [S1,S2,New,S4,S5,S6,S7,S8,S9]) :- test (Old).
replace(pair(4, New),
        [S1,S2,S3,Old,S5,S6,S7,S8,S9],
        [S1,S2,S3,New,S5,S6,S7,S8,S9]) :- test (Old).
replace(pair(5, New),
        [S1,S2,S3,S4,Old,S6,S7,S8,S9],
        [S1,S2,S3,S4,New,S6,S7,S8,S9]) :- test (Old).
replace(pair(6, New),
        [S1,S2,S3,S4,S5,Old,S7,S8,S9],
        [S1,S2,S3,S4,S5,New,S7,S8,S9]) :- test (Old).
replace(pair(7, New),
        [S1,S2,S3,S4,S5,S6,Old,S8,S9],
        [S1,S2,S3,S4,S5,S6,New,S8,S9]) :- test (Old).
replace(pair(8, New),
        [S1,S2,S3,S4,S5,S6,S7,Old,S9],
        [S1,S2,S3,S4,S5,S6,S7,New,S9]) :- test (Old).
replace(pair(9, New),
        [S1,S2,S3,S4,S5,S6,S7,S8,Old],
        [S1,S2,S3,S4,S5,S6,S7,S8,New]) :- test (Old).
```

Non–recursive replace used by P11.

E.3 Other tasks

We included task specifications and prototypes for some general-purpose tasks that might appear in many programs. One of these, *append/3*, was identified in the students' programs. We include that task and prototype definitions for this task.

Table E–52: Other Tasks: *append/3*

Used by:	Sub-tasks:	`Prolog Code` or called sub-tasks
P1 P5 P7	$e(Y, X)$	`X = Y`
	$f(V, L, X)$	`X = [V|L]`
	$g(Y, X)$	`X = Y`

Append two lists, to create a third list.

Table E–53: Other Prototypes: SCHEMA_A

Prototype:	SCHEMA_A
Top predicate:	p/3
Prototype functors:	p/3

```
p([], L, A) :-  e (L, A).
p([H|T], L, A) :-
            p(T, L, S),
            g (H, V),
            f (V, S, A).
```

List recursion, corresponding to Gegg-Harrison's Schema A.
Used by P1, P5, P7 for *append/3*. This prototype may also be used for other simple recursions, e.g. *reverse/3*, but not by these students.

Detecting Design Decisions About Data Structures in Prolog Programs

DIANA BENTAL
Department of Artificial Intelligence
University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, UK
E-Mail: diana@aisb.ed.ac.uk

Abstract:

The tutoring of intermediate Prolog programmers is supported by reconstructing and critiquing the design decisions the student has made. We describe an empirical study of the design decisions made by intermediate programmers, concentrating on decisions that relate to the construction of data structures and how these affect the overall design of the program and its control structures. We explore the use of type inference for the automated recognition of decisions about data structures. We describe an implementation of a type inference system for design critiquing, we discuss its strengths and weaknesses and we describe its role in an architecture to infer student's intentions about data structures.

1 Introduction

The programs that intermediate students write give rise to different problems from those written by novices. Intermediate programming exercises are exercises in many aspects of design: in interpreting problem specifications, in breaking problems down into their constituent parts, in choosing data representations and in choosing algorithms. If we are to tutor intermediate students about how to program we must be able to recognise design decisions and critique them.

Exercises for beginners focus on canonical examples of a few aspects of the language. The exercises typically include detailed specifications which are given in terms of the programming language and programming concepts. Students are likely to try to solve the right problem and the choices of solution strategy are very limited. At this level, Proust has been quite successful as a means of detecting and critiquing a range of semantic errors in Pascal programs (Johnson & Soloway, 1984).

When we consider how to tutor intermediate students automatically, the problems become much harder. The space of possible designs for intermediate programs is large. There are many levels of decision which create the program and translate earlier decisions into code. At each level, decisions are made both about the structure of the program – i.e. the code that is written – and about its behaviour – i.e. the data structures that are created and manipulated. Problem specifications are given in terms of a problem domain, which leaves the student with many decisions about how to map from the problem domain into a programming language (Kant, 1985). The specification may also be incomplete even in terms of the problem domain, leaving the student to decide exactly what behaviour is required from their program.

Each design decision affects later decisions, sometimes in a most unfortunate way. Spohrer observes that some approaches to solving a problem tend to lead to incomplete or incorrect solutions whereas other approaches are more likely to lead to correct solutions (Spohrer *et al*, 1985). We have found that decisions about data representations are fundamental to the good design of code. Choosing the right data representation can help students to write code that is more likely to be correct, efficient and readable.

Observations of experienced software designers have shown that when they design a system, they form a sequence of mental models of the system. Each model in the sequence is capable of simulating the behaviour of that system (Adelson & Soloway, 1985). These mental simulations allow the designer to check that the mechanism really will exhibit the desired behaviour. Simulation of the complete model at a given level of abstraction reveals interactions between decisions made about components of the model at that level of abstraction, interactions which may not be obvious when each component is considered in isolation.

Novice software designers differ considerably from experts in the way that they design programs. Novice software designers experience difficulty in constructing models in which they can mentally simulate the effect of their design decisions, whether as a result of a lack of general software design skills (Adelson *et al*, 1985) or as a lack of design skills in a particular problem domain (Adelson & Soloway, 1985).

In this paper, we describe an analysis of Prolog programs which demonstrates that there is a considerable need for a critiquing system that considers the choice of data structures. We then go on to outline the way in which we have sought to take advantage of type inference, and assess the strengths and weaknesses of this technique.

2 An Empirical Study of Design Decisions

We have studied the design decisions made by students writing a Prolog program to play noughts and crosses against a human opponent. These programs were written by twelve MSc students in the seventh week of a nine week Prolog course. The programs typically consisted of between 150 and 300 lines of code.

An initial analysis of eight programs identified the objects in the game that were to be represented, the data structures that the students used to represent objects in the game and the task structures that manipulate the objects. We then expanded the study to include the remaining four programs. We used this study set of twelve programs to relate the data structures chosen to represent different game objects and the ease with which those data structures can be manipulated for various different tasks.

2.1 The Problem Specification

The problem specification was given partly in terms of the problem domain and partly in terms of Prolog code. The program was to maintain, display and control the state of the game, to accept the user's input and to play correctly itself. All the students were required to use a common top-level framework for their code. The top two predicates were specified as being generic for two-player games and were taken from (Sterling & Shapiro, 1989). These predicates called half a dozen other named predicates which the students supplied.

The specification was incomplete in various ways. In terms of the problem domain, the level of skill required from the computer player was not specified. The program was required to display the board to the user and to obtain moves from the user, but no formats were specified. In programming terms, the students had to devise algorithms and representations for many of the tasks and entities.

2.2 Data Structure Decisions

The variables manipulated by the predicates were required to represent particular entities in the game (e.g. the state of the game, the next move) but the details of the content and format of these variables were left to the students. These decisions had to be made in the context of both the problem domain, of programming in general and of the Prolog programming language in particular.

We now discuss the data structure decisions that are made in each context, the ways in which decisions about data structures in each context affect the design of the control structures and the ways in which decisions in different contexts interact. We illustrate these issues through one example, the representation of the game state, that affects all of the major tasks in the game-playing program.

The problem specification required that the state of the game must be represented explicitly as a data structure. In the context of the problem domain, all of the students decided that for this game they

A *list of squares* is a list with nine elements. Each element represents a square in the board, ordered in rows from left to right. Each square is an atom.

```
[empty, empty, empty,
 empty, x,     empty,
 empty, empty, o     ]
```

The centre square and one corner are taken.

Figure 1: Board representation: List of Squares

A *list of lines* is a list with eight elements. Each element represents one of the lines (three rows, three columns and two diagonals). Each line is itself a list which contains three squares.

```
[[1, 2, 3], [4, x, 6], [7, 8, o],
 [1, 4, 7], [2, x, 8], [3, 6, o],
 [1, 5, o], [3, x, 7]]
```

Empty squares are represented by integers that reflect their position. The centre square (number 5) and one corner (number 9) are taken.

Figure 2: Board representation: List of Lines

would represent the state of the game by the contents of the board, rather than for example by collecting the moves made so far.

Four different representations of the board are shown in Figures 1, 2, 3, and 4. In terms of the problem domain, the students can choose whether to represent the nine individual squares or the eight lines in the board. Table 1 shows the trade-off in this decision in terms of the complexity of different tasks. Some data structures are easily manipulated for some purposes but not others. The task of finding complete and partial lines can be greatly simplified by representing each of the eight lines explicitly. However, the task of putting a token into the right place in the board is more difficult in this representation, for each square appears in more than one line.

Students who explicitly represented the lines in the board had a further choice of whether to distinguish explicitly between rows, columns and diagonals (Figure 4) or whether to make no distinction between the different sorts of line (Figure 3). It is often useful to make explicit the distinctions between different sorts of entity, but in fact almost all of the processing treats all the lines uniformly. As a result, students who did made the distinction explicit typically had more complex and error-prone programs than those who did not.

We now consider decisions in terms of general programming techniques and their implementation in Prolog. One significant decision in representing the board and other data structures is whether to use a

Table 1: Comparison of board representations for different tasks

We compare how easy it is to implement two different tasks using four different board representations. The utility of the representation for the task reflects the conciseness, readability and efficiency of the best implementation that uses that representation.

Board Representation	Task	Utility
List of Squares	Recognise a good move	Hard
	Update the board	Easy
List of Lines	Recognise a good move	Easy
	Update the board	Hard
Term of Lines	Recognise a good move	Hard
	Update the board	Hard
Nested Lines	Recognise a good move	OK
	Update the board	Hard

A *term of lines* structure is a term with eight subterms. Each subterm represents one of the lines (three rows, three columns and three diagonals). Each subterm contains three squares.

```
board(line(one, two,  three), line(four,  x, six),   line(seven, eight, o),
      line(one, four, seven), line(two,   x, eight), line(three, six, o),
      line(one,   x,    o),   line(three, x, seven))
```

Empty squares are Prolog atoms. The centre square and one corner are taken.

Figure 3: Board representation: Term of Lines

A *nested lines* structure is a term with three subterms. One subterm represents rows, the second columns, the third diagonals. Each subterm contains the appropriate lines, which in turn contain squares.

```
board(row(line(one, two, three),     line(four, x, six),  line(seven, eight, o)),
      column(line(one, four, seven), line(two, x, eight), line(three, six,  o)),
      diagonal(line(one, x, o),      line(three, x, seven)))
```

Empty squares are Prolog atoms. The centre square and one corner are taken.

Figure 4: Board representation: Nested Lines

record (with a fixed number of entries which may be of different types) or a recursive data structure (with a variable number of entries, all of the same type). The basic Prolog data structure is a term with a fixed number of arguments which are accessed by pattern matching. This makes an obvious record structure (e.g. the boards in Figures 3 and 4). The arguments can also be complex terms, including similar terms nested indefinitely deeply, and so it is possible to build recursive data structures such as lists and trees from terms.

Prolog also supports two special notations for lists. Both of these notations disguise the underlying implementation, which is a term with a fixed number of arguments indefinitely nested. One notation `[H|T]` supports the concept of a list as a linear data structure of indefinite size, whose elements are accessible by recursion. The other notation `[a,b,c]` also supports the concept of a list as a linear data structure, but it forces the list to be of fixed size and it makes its elements accessible by pattern matching as well as by recursion (e.g. Figures 1 and 2). The two notations can be combined for a list whose initial entries are in some fixed pattern but whose tail is not specified.

Prolog syntax therefore neatly blurs the distinctions between fixed and variable sized structures, and between access by position and access by recursion. It is not surprising that some students were confused.

Some students tried to manipulate Prolog terms as if they were also linear data structures of indefinite size. This is possible in Prolog (using the built-in predicate `arg/3`), and it is useful on occasion, but it is awkward and inefficient. In the noughts and crosses program it is never appropriate to do this and a list should be used instead. Students who chose to use a term to represent the board (Figures 3, 4) were led either to write large amounts of repetitive pattern-matching code to access the different elements and then treat them in the same way, or else to use recursion with the verbose and inefficient `arg/3` predicate.

2.3 Summary of Data Analysis

Our analysis of students' data structure decisions shows that each choice of data representation interacts with the choices of data representation for their components and also with the quality of design of the code that manipulates those data structures. Design decisions for widely used data structures may involve trade-offs for different tasks.

We find that students have some particular difficulties in understanding when particular data representations should be used. A poor decision can make the implementation of the rest of the code more difficult.

3 A Mechanism for Recognising Data Structures

We now turn to the problem of recognising and critiquing the student's design decisions that relate to the choice of data structure. In this section we outline our general requirements for recognising data structures, we describe a technique, *type inference*, which is applicable to the problem and we discuss the suitability and limitations of type inference.

We require an abstract language in which to describe the data structures that have been used in the program, and a mechanism which will infer what data structures have been used and express the results in that abstract language. We use an abstract interpretation approach with polymorphic data types as the language and polymorphic type inference as the mechanism.

Abstract interpretation runs the program, but not under the usual interpreter (Abramsky & Hankin, 1987). Instead the program is run under an interpreter that uses data types instead of data values, and that uses inputs and produce outputs that are not the conventional inputs and outputs but type abstractions of them. By using different languages and different interpreters, we can obtain different views of the program and different levels of abstraction in our descriptions.

Type mechanisms have been used to increase the run-time efficiency of programs, to provide simple and powerful mechanisms for data abstraction and to help programmers write programs whose use of data structures is consistent and are therefore more correct and more robust. Type inference has also been used in intelligent tutoring systems for computer programming, but with a rather different focus. Looi has used type inference for small programs with a limited range of data types in order to facilitate plan-based matching (Looi, 1988), rather than to identify and critique decisions about data structures.

An alternative is a plan-based approach which compares the structure of the code to canonical code which creates particular data structures. This has been applied successfully to programs which exhibit little variation in implementation (Letovsky, 1988) but it cannot easily be applied to implementations whose plan structure is unfamiliar, and we do not pursue it further.

We have implemented a type inference mechanism which is given declarations for data types and for built-in predicates and works out the type of each variable within each of the user's predicates. This approach is based on Mycroft and O'Keefe's polymorphic type system for Prolog (Mycroft & O'Keefe, 1984) and on those used for polymorphic typed languages such as ML.

Type mechanisms have generally been used by program designers who supply the necessary type declarations. For instance, Mycroft and O'Keefe's type inference system requires type declarations for all predicates in the program and for all data structures it manipulates. By contrast, we are dealing with completed programs for which the student programmer has written neither data structure declarations nor predicate type declarations. Our task is not to check that a given specification for types is correct for a particular program but to infer what the specification should be for that program. Some parts of the type specification are general and can be supplied in advance, others can be inferred from the program, and still others can only be predicted but with no certainty of success. In the following sections we describe our inference mechanism and its use in recognising data structure representations.

3.1 Implementation

We illustrate the inference process using the example in Figure 5. The objective of the type inference system is to derive a **predicate_type** declaration for the predicate **find_empty_square** in terms of the declared data types (**list**, **square** and **integer**) and the declared **predicate_type** for **is**. For illustration, we assume that data type declarations are supplied for all data types, including the domain-specific type **square**. Section 3.2 discusses our mechanism for the situation in which we cannot predict the declarations that are needed for domain-specific data types.

The first data type declaration in Figure 5 states that a list of elements is either empty (`[]` in the Prolog program) or it consists of an element attached to a list of elements (`[_|_]` in Prolog is analogous to the **cons** operation in LISP). The type of the elements in a list is a type variable, so that a list may contain elements of any data type but all the elements of the list are constrained to be of the same type. This constraint on lists is more specific than the Prolog interpreter demands, and it causes difficulty in the type analysis where a student has used a list instead of a term to implement a record structure whose entries are of different types, but it is typical of the good use of Prolog recommended in e.g. (O'Keefe, 1990).

Given the declarations:

```
:- type list(A) --> [] ; [A|list(A)].
:- type square --> x ; o ; empty.
:- type integer --> integer + integer.

:- predicate_type(integer is integer).
```

and the code:

```
find_empty_square([empty|_], 1).
find_empty_square([_|Rest], Position) :-
        find_empty_square(Rest, P1),
        Position is P1 + 1.
```

We infer:

```
find_empty_square(list(square), integer))
```

i.e. that the board is a list of squares.

Figure 5: Example of Type Inference

The **predicate_type** declaration states that the built-in Prolog predicate **is** has two arguments, both of which must be integers.

In brief, the mechanism works as follows. To infer the argument types of a predicate, the type inference mechanism processes each clause in turn. Bindings are maintained to appropriate types or type variables for each argument in the head of the clause and for each variable referenced in the head or body of the clause.

For each call to another predicate in the body of the clause, we derive the types that the called predicate expects. This means either looking up an existing **predicate_type** declaration or making another call to the same type inference mechanism. In the case where the predicate call is itself recursive, as is the first call in the second clause of **find_empty_square**, we maintain a stack of the type information for the predicates we have analysed so far. (This information is restricted to the current clause in the current predicate, not the whole predicate, which is a weakness in our mechanism but simplifies the processing).

Having obtained the type of the called predicate, we obtain the types used in the call, compare the two and add any further information to the variable bindings list. This is repeated for the remaining calls in that clause. When types have been derived for each clause they are combined to give the overall type of the predicate.

An extra mechanism must be supplied to deal with numbers, and with any other types whose values cannot be enumerated explicitly. This is easily done for types like integers which are built in to Prolog. If a programmer were to define such a type then a special-purpose extension would be needed to the type checker for each such type, but this is rare in practice.

Figure 5 shows, by necessity, a simple example to illustrate the inference mechanism. Our system is able to infer types for student programs whose predicates are are deeply nested, recursive and mutually recursive. This mechanism is unable to derive useful information for parts of some programs, in particular for calls to some Prolog meta-predicates (not discussed further) and for parts of a program that are typed inconsistently.

3.2 Domain-Specific Data Structures

It is possible to derive the predicate type declarations for the predicates written by the student, but only if the type inference mechanism is supplied with the data type declarations. Unfortunately there is no general mechanism which is able to derive the necessary data type declarations.

It easy to supply the data type declarations for general data structures such as lists or integers. A problem arises when students use data structures that are specific to the domain or chosen idiosyncratically by the student. In the example in Figure 5 we inspected the program to find the values that the student used to represent the **square** and we supplied the appropriate type declaration.

For these data structures, we can perform type inference using only their sizes and not their names. For instance, we may infer that the student has represented a square by an atom without needing to

know that the actual names are `x`, `o` and `empty`. These anonymous types (atom, singleton, pair, triple etc) need not be declared but are built in to the type inference mechanism and they are recognised when the data structure does not match any of the explicit declarations. This allows us to recognise data structures at the most specific level of abstraction, as has been done with recognising code structures (Greer *et al*, 1989; Corbett *et al*, 1988).

3.3 Recognising Design Decisions in the Problem Domain

Type inference does not in itself solve the problem of mapping between the students' intentions in terms of the problem domain, and the students' implementation in terms of the programming language. Some data types (such as lists or integers) may be used in several different contexts to represent different entities, and the type inference system does not distinguish between these different uses.

We treat type inference as one component of a larger system for recognising design decisions to do with data structures and algorithms (Bental, 1992). Information about data types is passed to a knowledge–based system which combines this information with domain–specific information about plausible representations for domain entities and the relations between them.

In this system, each plausible representation for a domain entity (such as the board) is described by the data type that corresponds to that representation and also by optional constraints on which objects must appear as parts of that object. Each example in Figures 1 to 4 is a different board representation. For instance, the *list of lines* representation for the board (Figure 2) is constrained to be a list of elements. The type of the elements is not specified but elements are constrained to be entities recognised as *lines*. Lines are in turn constrained to be either lists (as in the figure) or else triples of entities that are of an appropriate type to be *squares*.

To identify the representation of an object, we recursively check that the inferred data type of the game object and its parts matches one of the data types for that object and the appropriate parts. We use expectations derived from the problem specification to constrain the matching of possible domain entities.

4 Discussion

Polymorphic type inference provides a language with which to describe and identify the data structures that are used and created by predicates. We use it to describe the behaviour of Prolog programs and the intentions of students in writing those programs.

Generality in data structures is expressed along two dimensions. Type variables enable us to distinguish between, say, a procedure which creates or manipulates a list of integers and a procedure which works for all lists regardless of the types of their elements. This generality is similar to the hierarchical abstraction that is used to describe functions in the Lisp Tutor (Corbett *et al*, 1988). Anonymous types allow us to identify a data type in less detail than types which match a specific declaration. This generality is similar to the abstraction used in the SCENT tutor (Greer *et al*, 1989), which abstracts from to a program with a specific syntactic structure to a general Lisp program.

The language has some limitations. It cannot express all the information that we might like to derive about data structures. For example, data structures must be treated either as having finite size or as being infinite. This means that we cannot use this mechanism to distinguish between lists of different lengths which are used for different purposes.

Prolog is not a strongly typed language, and so students can write and successfully run programs that are inconsistently typed according to our mechanism. A common example is where a student has used a Prolog list instead of a term to implement a record. In some cases, we may view this as a limitation on the ability of type inference systems in general (e.g. (Dietrich & Hagl, 1988; Fruhwirth, 1988)). In other cases, the observation that a data type is used inconsistently can itself be a useful comment on the design (Mycroft & O'Keefe, 1984). Even when part of the implementation uses types inconsistently, our system is able to obtain some information about the data type that the student intended from other parts of the program.

5 Conclusions

The choice of appropriate data structures is important for the writing of well designed code. Data structure decisions are made in terms of the problem domain as well as the implementation language. We have found that intermediate programmers often have difficulty in choosing appropriate data structures, and this difficulty has a major effect on the quality of the rest of the design.

We have found that a language of abstract data types together with a flexible type inference mechanism can usefully capture some of these decisions about data structure design at different levels of abstraction. Such a mechanism forms a useful component of a system that can identify decisions made in the problem domain and in the implementation language.

References

Abramsky, S. & Hankin, C. (editors). (1987). *Abstract Interpretation of Declarative Languages.* Ellis Horwood.

Adelson, B., Littman, D., Ehrlich, K., Black, J. & Soloway, D. (1985) Novice-expert differences in software design. In *Proceeding of the Conference on Human-Computer Interaction — INTERACT 84*, pages 473–478.

Adelson, B. & Soloway, E. (1985). The role of domain experience in software design. *IEEE Transactions on Software Engineering*, 11:1351–1360.

Bental, D. (1992). Using clausal join and clausal split to recognise language–specific programming design decisions. In *Proceedings of the First AAAI Workshop on AI and Automated Program Understanding*, pages 37–41.

Corbett, A.T., Anderson, J.R. & Patterson, E.G. (1988). Student modeling and tutoring flexibility in the Lisp intelligent tutoring system. In *Proceedings of the International Conference on Intelligent Tutoring Systems*, pages 83–106.

Dietrich, R. & Hagl, F. (1988). A polymorphic type system with subtypes for Prolog. In H. Ganziger, editor, *ESOP-88: Proceedings of the Second European Symposium on Programming*, pages 79–93.

Fruhwirth, T.M. (1988). Type inference by program transformation and partial evaluation. In *Proceedings of META 88*, pages 199–212.

Greer, J.E., Mark, M.A. & McCalla, G.I. (1989). Incorporating granularity-based recognition into SCENT. In *Proceedings of the 4th International Conference on AI and Education*, pages 107–115, Amsterdam.

Johnson, W.L. & Soloway, E. (1984). Intention-based diagnosis of programming errors. In *Proceedings of the National Conference on Artificial Intelligence*, pages 162–168.

Kant, E. (1985) Understanding and automating algorithm design. *IEEE Transactions on Software Engineering*, 11:1361–1373, 1985.

O'Keefe, R.A. (1990). *The Craft of Prolog.* Cambridge, Mass: MIT Press.

Letovsky, S.I. (1988). *Plan Analysis of Programs.* PhD thesis, Department of Computer Science, Yale University.

Looi, C-K. (1988). *Automatic Program Analysis in a Prolog Intelligent Teaching System.* PhD thesis, Department of Artificial Intelligence, University of Edinburgh.

Mycroft, A. & O'Keefe, R.A. (1984). A polymorphic type system for Prolog. *Artificial Intelligence*, 23:195–307.

Spohrer, J.C., Soloway, E. & Pope, E. (1985). A goal/plan analysis of buggy Pascal programs. Technical report, Department of Computer Science, Yale University.

Sterling, L.S. & Shapiro, E. (1986). *The Art of Prolog: Advanced Programming Techniques.* Cambridge, Mass: MIT Press.

Acknowledgements

This work has been supported by a SERC studentship. Thanks are due to Dr. P. Brna and Dr. P. M. Ross for supervision, and to Richard Cox and Dave Berry for comments.

Using Clausal Join and Clausal Split to Recognise Language–Specific Programming Design Decisions

Diana Bental*
Department of Artificial Intelligence, University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN, U.K.

diana@aisb.ed.ac.uk

Abstract

In this paper we describe part of our research on recognising design decisions in intermediate students' Prolog programs. We outline a methodology for program design and describe a mechanism called *clausal join* that supports this methodology. We describe how we have used the clausal join mechanism in program analysis and how we have developed an inverse mechanism called *clausal split*.

1 Introduction

We are investigating software analysis methods which contribute to the tutoring of intermediate students in computer programming. To do this we must recognise and critique the design and implementation decisions that intermediate students have made in writing medium–sized programs.

The problems of tutoring intermediate students are on a larger scale than those for novices. Intermediate programming exercises are exercises in design: in interpreting problem specifications, in breaking problems down into their constituent parts, in choosing data representations and algorithms, both language independent and language dependent [2]. We have discussed the levels at which design decisions are made and described the analysis of data representations in [1].

We have also devised a representation for language-specific algorithms which combines representations used in automated software analysis (especially [4; 8]) and representations used in program synthesis and design [6; 3] and from a design method suggested by O'Keefe [9]. The rest of this paper describes one part of this, in which we have taken a technique called clausal join that has been used for automated program synthesis, and applied and extended it for use in software analysis.

2 Clausal Join: A Method for Program Synthesis

Lakhotia and Sterling have described a method for automatically composing programs with similar control structures [7]. They describe this work in the context of small programs which traverse simple data structures such as trees and lists, and also of complex programs such as interpreters which perform different operations on more complex data structures. Their goal is to support the modular development of complex software.

Lakhotia and Sterling distinguish between *skeletons*, which describe the basic control flow of the procedure, and *techniques*, which may be applied to skeletons to produce *enhancements*. Some enhancements, called *mutations*, change the control flow of the procedure. Others do not, and these are called *extensions*. For example a mutation of a tree traversal procedure might search the tree for a leaf of a particular value and stop when such a leaf is found, whereas an extension might count all its leaves. Sterling and Lakhotia observe that procedures which are extensions of the same skeletons may be composed to give single procedures which correctly perform various functions 'in parallel'. Clausal join is the method that produces these compositions.

They conjecture that a useful program development method for Prolog would give programmers access to a range of re-usable skeletons which they could extend appropriately, plus the clausal join algorithms for composing the extensions.

2.1 Example of a Simple Clausal Join

There are two variants of clausal join. The central algorithm for combining clauses is the same for both, the only difference being how clauses are selected for merging. The one–one join merges each clause of one component with just one appropriate clause of the other. The one–one join is appropriately used for components which traverse the same instance of a data structure, so that for instance clauses that deal with leaves of a tree are joined and clauses that deal with branches are joined. The other join, procedural join, merges each clause of one component with every clause of the other. It is used to process different instances of the same data structure which may be of different sizes. We now concentrate only on the one–one clausal join.

The clausal join procedure is described in detail in [7]. We briefly illustrate the clausal join procedure using an example of two procedures in Figure 1, `count` and `sum`, which compute the number and sum of elements in a list of integers. Their composition might be used as part of a procedure to compute the mean.

A clausal join is controlled by a join expression

*Supported by a SERC studentship. Thanks are also due to Dr. P. Brna and Dr. P. M. Ross for supervision, and to Dr. P. Brna for comments.

```
count([], 0).
count([H|T], Count) :- count(T, C1),
                       Count is C1+1.

sum([], 0).
sum([H|T], Sum) :- sum(T, S1),
                   Sum is S1+H.
```

Figure 1: Component Procedures

```
count_and_sum(List, Count, Sum) :-
    count(List, Count), sum(List, Sum)
```

Figure 2: Join Expression

which states how the composed procedure relates to its components, and in particular, how the arguments to the procedures are merged. Our example uses the join expression in Figure 2, written in a standard Prolog notation.

The join operation first unifies the RHS of the join expression with the heads of the component clauses, making appropriate variable substitutions. The head of the composed procedure is the resulting LHS of the join expression. Thus, for the first clause we have

```
count_and_sum([], 0, 0) :-
    count([], 0), sum([], 0)

count_and_sum([], 0, 0) :- ...
```

For the second clause of the combined procedure, we have

```
count_and_sum([H|T], Count, Sum) :-
    count([H|T], Count), sum([H|T], Sum)

count_and_sum([H|T], Count, Sum) :- ...
```

We then create the body of the composed clauses by unfolding the bodies of the component clauses. The body of the first clause of both procedures is empty, leaving the first clause of the composed procedure as:

```
count_and_sum([], 0, 0).
```

For the second clause, unfolding results in:

```
count_and_sum([H|T], Count, Sum) :-
    count(T, C1), Count is C1+1,
    sum(T, S1), Sum is S1+H.
```

We now fold the calls to sum and count together using the join expression. The composed procedure that results is given in Figure 3.

The join expression indicates that the first arguments of count and sum are to be merged into the first argument of count_and_sum, and that the second arguments of both procedures are to be transferred as they are. The heads of the clauses and the recursive calls are processed by the same join expression. The auxiliary calls are transferred directly from the component procedures into the composed procedure, with appropriate variable bindings.

```
count_and_sum([], 0, 0).
count_and_sum([H|T], Count, Sum) :-
    count_and_sum(T, C1, S1),
    Sum is S1 + H,
    Count is C1 + 1.
```

Figure 3: Composed Procedure

The composed procedure is more efficient than calling the two component procedures because it traverses the list only once. Clausal join can also applied to the more efficient tail recursive versions of sum and count to produce a tail recursive composition.

We now consider how the clausal join technique could be used in program analysis and the need for its inverse, clausal split.

3 Program Analysis

In our analysis of student's programs, we wish to identify the purpose of a piece of code and also to comment on its design. Even for small programs the programmer must make many different decisions, and so we cannot explicitly list all of the possible implementations of the students' code [5]. Instead we need a more abstract way to describe, recognise and critique programs. Skeletons, techniques and enhancements provide part of such an abstract description.

In our study of programs written by intermediate programmers we have found that badly designed programs may arise from the choice of the wrong control skeleton for the task or from applying the wrong technique to it. Our objective is to design an automated process which can recognise which skeletons have been used in a design, and which techniques have been applied to it.

Our process begins by taking an analysis–by–synthesis approach [10] in which we apply techniques to skeletons and match the resulting code against the student's code. A student's implementation of a procedure may either be serial (e.g. to calculate the mean of a list of numbers, count the elements in a list and then compute the sum) or composed (e.g. traverse the list once, counting and summing the elements.) We use the clausal join on the system's generated implementations to generate further composed implementations that can be matched to the students' code.

A purely analysis–by–synthesis approach does not work well, however, if one of the components of the students' composed procedure is novel or otherwise cannot be recognised. In this case we may wish to work backwards from the students' composed procedure to its components, to identify the skeletons and techniques in those components for which this is possible and to isolate the other components. In order to isolate these components it is useful to have a *split* operation, the inverse of the join, which would split a joined procedure into its component parts.

4 An Algorithm for Clausal Split

We now outline the algorithm for splitting a composed clause into its components. The inputs to the process are the composed procedure and the join expression.

The join expression does not indicate which auxiliary calls belong in which component. This problem does not arise in the join, in which all auxiliary calls are simply transferred from the components into the composed procedure. We can use data flow between variables to decide where each auxiliary call belongs.

We cannot split procedures which contain dataflow dependencies between variables unless those dependencies have been captured by the join expression. This means that, as for clausal join, all variables on the RHS of the join expression must appear on the LHS. It also means that if the join expression indicates that two variables belong in two different components but a single auxiliary call refers to both variables, then we cannot tell what to do with the auxiliary call and the split cannot proceed. A (somewhat contrived) example in which a clause could not be split would be if the second clause of `count_and_sum` contained an extra call:

```
count_and_sum([H|T], Count, Sum) :-
    count_and_sum(T, C1, S1),
    Sum is S1 + H,
    Count is C1 + 1,
    Total is Sum + Count.
```

A more sophisticated approach would be: if there existed a join expression for the auxiliary call then recursively split both the auxiliary call and the procedure that is called, but we do not explore this further.

Strictly speaking, where a join expression joins more than two components we may need to distinguish between variables that are shared between all the components, and variables which are shared between different subsets of components. This is a straightforward extension which we do not discuss further.

4.1 The algorithm

The algorithm proceeds one clause at a time through the composed procedure. It splits each clause of the composed procedure into one corresponding clause for each of the components. Each composed clause is split independently, and the matches between variables that are created in splitting one composed clause do not affect the matches for other composed clauses. We illustrate the process by splitting the second clause of the procedure `count_and_sum` in Figure 3 using the same join expression (Figure 2).

Obtain the heads of the split clauses First we match the LHS of the join expression to the head of the composite clause and transfer corresponding arguments to the RHS of the join expression. (Both match and transfer are done by unification.) This gives us the arguments in the heads of all the components.

```
count_and_sum([H|T], Count, Sum) :-
    count([H|T], Count), sum([H|T], Sum)

count([H|T], Count) :- ....
sum([H|T], Sum) :- .....
```

We enter all variables that appear in more than one call on the RHS of the join expression into the set of *shared* variables and all variables that appear in only one call on the RHS of the join expression to the set of *private* variables for that call. In our example, the sets are:

Private to `count`:	`Count`
Private to `sum`:	`Sum`
Shared:	`H, T`

Deal with recursive calls If there are any recursive calls in the body of the clause, we split each recursive call by matching the composite call to the LHS of the join expression and deriving the components by transferring arguments to the RHS, exactly as for the heads.

```
count_and_sum(T, C1, S1) :-
    count(T, C1), sum(T, S1)

count([H|T], Count) :- ... count(T, C1) ...
sum([H|T], Sum) :-  ... sum(T, S1) ...
```

We extend the sets of private variables so that any variable that is included in a split recursive call but is not a shared variable, is assumed to be private to that split clause and is added to the set of private variables for that clause. In our example, the sets are:

Private to `count`:	`Count, C1`
Private to `sum`:	`Sum, S1`
Shared:	`H, T`

Decide which auxiliary calls are to be included We have now created one procedure for each component with an appropriate head and appropriate recursive calls. We also have sets of the variables we know so far are private to each component and a set of shared variables. We now decide which auxiliary calls should be included in which components.

We repeatedly pass through the auxiliary calls. If a call refers to any variables that are private to a component, then we add the call to the appropriate component procedure, and we also add any new variables in the call to the set of private variables for that component. We also check that none of the other variables in the call are private to any other component, as this would mean that a correct split is not possible. We repeat this procedure until there is a pass for which no more auxiliary calls have been added to any component procedure.

Any auxiliary calls that remain at the end of this process do not contain any private variables. We assume that such calls are relevant to all the components, since they do not include any dataflow that is private to a single component, and so they are copied into every component procedure.

The procedure `count_and_sum` contains two auxiliary calls. `Count=C1+1` refers only to variables private to `count`, and is therefore included in the body

of `count`. `Sum=S1+H` refers to variables private to `sum` and to an explicitly shared variable and is therefore included in the body of `sum`. No further extensions are made to the sets of private variables. The resulting clauses are, as expected:

```
count([H|T], Count) :- count(T, C1),
                       Count is C1+1.
sum([H|T], Sum) :-  sum(T, S1),
                    Sum is S1+H.
```

Continue This process is repeated for all the clauses in the composed procedure and results in the procedures in Figure 1.

4.2 Results

The clausal split algorithm works well on the `sumcount/3` and `tree` examples given in Lakhotia and Sterling [7]. It gives splits with correct auxiliaries. It has also been tested successfully on examples in which the dataflow between auxiliary calls is complex.

During program synthesis, the join expressions can be created and applied as the program is created and the decision about which join expression to apply can be made by the programmer. During program analysis, we must decide which join expression is appropriate. We are currently exploring this problem, using a knowledge based approach.

5 Conclusion

We have designed and implemented an algorithm which successfully inverts the join procedure and derives the components of a composed procedure from the procedure and the join expression. Given a notion of independent dataflow within the component procedures it decides which auxiliary calls belong in which component

Preliminary results suggest that the skeletons, techniques and compositions used in program synthesis are also useful for program analysis and design critiquing for Prolog programs.

References

[1] Diana Bental. Critiquing design decisions in prolog programs. Submitted to *The 10th European Conference on Artificial Intelligence*, 1992.

[2] Paul Brna, Alan Bundy, Tony Dodd, Marc Eisentadt, Chee-Kit Looi, Helen Pain, Dave Robertson, Barbara Smith, and Maarten van Someren. Prolog programming techniques. *Instructional Science*, 20(2):111–133, 1991.

[3] Alan Bundy. Proposal for a recursive techniques editor for Prolog. Technical Report 394, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, UK, 1988.

[4] Timothy S. Gegg-Harrison. Basic Prolog schemata. Technical Report CS-1989-20, Department of Computer Science, Duke University, 1989.

[5] W.L. Johnson and E. Soloway. Intention-based diagnosis of programming errors. In *Proceedings of the National Conference on Artificial Intelligence*, pages 162–168, 1984.

[6] Marc Kirschenbaum, Arun Lakhotia, and Leon S. Sterling. Skeletons and techniques for Prolog programs. Technical report, Case Western Reserve University Center for Automation and Intelligent Systems Research, Cleveland, Ohio, 1989.

[7] Arun Lakhotia and Leon S. Sterling. Composing recursive logic programs with clausal join. In *Proceedings of the Workshop on Partial and Mixed Computation*, 1987.

[8] Chee-Kit Looi. *Automatic Program Analysis in a Prolog Intelligent Teaching System*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, 1988.

[9] Richard A. O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, Mass, 1990.

[10] B.J. Reiser, J.R. Anderson, and R.G. Farrell. Dynamic student modelling in an intelligent tutor for LISP programming. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 8–14, 1985.